University of Alberta

Language Extensions for Multiple Code Inheritance in Java

by

Calvin Puihong Chan

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

0-494-08036-1

Your file *Votre référence*
ISBN:
Our file *Notre reference*
ISBN:

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

To my wife Connie and my son Colmon,
my parents Chark-wan and Ngan-yu Chan

Abstract

The object-oriented programming paradigm provides strong support for code re-use via inheritance mechanisms. Currently, Java is one of the most widely used objected-oriented programming languages, but Java supports only single code inheritance. Unfortunately, code repetition is sometimes unavoidable without some form of multiple code inheritance. Historically, multiple code inheritance mechanisms in programming languages have been problematic. This is due to the tight coupling of code and data layout, along with fundamental semantic issues with multiple data layout inheritance. Since Java's class mechanism is used for both inheriting code for methods (i.e., code-type) and inheriting state information (i.e., data-type), a mechanism for separating the code-type and data-type inheritance is necessary to solve the multiple code inheritance problem.

We describe the implementation of MCI-Java, an extension to Sun's Java 1.2.2 that separates code-types and data-types and thereby enables a multiple code inheritance mechanism. IBM's Jikes Compiler 1.15 is modified to provide compiler support for MCI-Java. At the source-code level, minor Java syntax changes are required for MCI-Java. At the Virtual Machine level, minor changes to the code loader and linker are required, but no change is made to the highly-optimized bytecode execution procedure.

Lastly, we empirically demonstrate that the MCI-Java Virtual Machine, and the modified IBM Jikes Compiler, do not degrade the performance of Java programs, when compared with the classic Java Virtual Machine and the original IBM Jikes Compiler.

Acknowledgements

I would like to express my sincere and cordial thanks to my supervisors, Professor Duane Szafron and Professor Paul Lu, for their invaluable input, continuous support and understanding especially when I was having a serious health problem. It would not be possible for me to complete this project without their guidance and support.

I would like to thank Maria Cutumisu for her patience and helpful advice every time when I encountered problems during my research, in particular, her work in refactoring the Java classes. Without her work, the task to demonstrate the effectiveness of MCI-Java would not be as easy.

I would like to acknowledge Dominique Parker for his help in running the Java demonstration package Java2D to test the compatibility of MCI-Java for unmodified Java programs.

Thanks also go to the Software Systems Research Group in the University of Alberta for maintaining a pleasant environment in the laboratory. They have been amiable to me since I joined the group.

I am appreciative of the contributions from my brother Pui-lam and my sisters – Maria and Ivy – to take care of my parents so that I can concentrate on my study.

I am grateful to my wife Connie for her love, encouragement and understanding throughout my study. It is her support that gave me the strength to complete my study. I am also owing to my son Colmon who deserved more care and love from me during my study.

Last but not the least, I am indebted to my parents Chark-wan and Ngan-yu for their love and support throughout my life. This thesis is for my entire family.

Table of Contents

1 Introduction 1
- 1.1 Overview . . . 1
- 1.2 Motivations . . . 3
- 1.3 Contributions . . . 5
- 1.4 Notation . . . 6
- 1.5 Organization . . . 6

2 Code Inheritance 7
- 2.1 Common code . . . 7
- 2.2 Multiple representation . . . 8
- 2.3 Overview of OOP languages . . . 11
  - 2.3.1 Separation . . . 11
  - 2.3.2 Code inheritance . . . 12
- 2.4 Summary . . . 13

3 Extensions to the Java Programming Language 15
- 3.1 Guidelines for changes . . . 15
- 3.2 Orthogonality in language design . . . 17
- 3.3 MCI-Java . . . 18
  - 3.3.1 Implementation . . . 18
  - 3.3.2 MultiSuper method invocation . . . 20
- 3.4 The semantics of multiple code inheritance in MCI-Java . . . 20
  - 3.4.1 Inheritance and visibility . . . 20
  - 3.4.2 Precedence of supertypes . . . 21
  - 3.4.3 Inheritance conflict and resolution . . . 22
  - 3.4.4 Inheritance suspension . . . 24
  - 3.4.5 Keywords `this` and `super` . . . 25
  - 3.4.6 Inheritance scenarios for MCI-Java . . . 26
- 3.5 The semantics of multisuper call in MCI-Java . . . 26
- 3.6 Summary . . . 27

4 Compiler Support 29
- 4.1 The extended *`.class`* file . . . 30
- 4.2 The grammar of MCI-Java . . . 32
- 4.3 Inheritance and conflict . . . 34
  - 4.3.1 The data structure - `ExpandedMethodTable` (EMT) . . . 34
  - 4.3.2 The revised EMT building algorithm . . . 36

4.3.3 Revised algorithm verification . . . 37
4.4 Bytecode generation . . . 39
4.5 Summary . . . 40

5 Virtual Machine Support **41**
5.1 The Java Virtual Machine (JVM) . . . 42
5.1.1 The run-time environment . . . 42
5.1.2 Dynamic per-class structures creation . . . 44
5.2 Class data verification . . . 44
5.3 Method tables . . . 45
5.3.1 `Method Block` . . . 46
5.3.2 `Method Table` (MT) . . . 46
5.3.3 `Virtual Method Table` (VMT) . . . 47
5.3.4 `Interface Method Table` (IMT) . . . 50
5.3.5 Setting offset values in `Method Blocks` . . . 53
5.4 Method lookup and invocation . . . 54
5.4.1 Overview . . . 54
5.4.2 Dereferencing the `ConstantPool` entry . . . 55
5.4.3 Method lookup . . . 56
5.4.4 The quicking process and code execution . . . 58
5.4.5 Multisuper call . . . 60
5.4.6 Example scenarios . . . 61
5.4.7 Recompilation of a supertype . . . 62
5.5 Summary . . . 64

6 Empirical evaluation **67**
6.1 Compatibility of MCI-Java for unmodified Java programs . . . 67
6.2 Correctness of MCI-Java generated code . . . 68
6.3 Refactoring legacy code . . . 68
6.3.1 Duplicate code promotion . . . 68
6.3.2 Static method promotion . . . 69
6.3.3 Prefix method promotion . . . 70
6.3.4 Super-suffix method promotion . . . 71
6.4 Refactoring the `java.io` package . . . 73
6.5 Performance of Jikes and MCI-Java VM . . . 77
6.5.1 Compiler test . . . 77
6.5.2 I/O test using refactored I/O *classes* . . . 78
6.6 Summary . . . 79

7 Conclusions **81**
7.1 Related work . . . 81
7.1.1 Mixins . . . 81
7.1.2 Traits . . . 82
7.1.3 Code in *interface* . . . 83
7.2 Summary of work . . . 84
7.3 Research contribution . . . 84

Bibliography **87**

A Abbreviations 91

Appendix 92

B Class File Structure 93

C The Grammar of MCI-Java 95
C.1 Introduction . . . 95
C.2 The Grammar of MCI-Java . . . 96

D Multiple Code Inheritance Scenarios 105
D.1 Single inheritance involving an *implementation* . . . 106
D.1.1 Scenarios 1 - 2 . . . 106
D.1.2 Scenario 3 . . . 106
D.1.3 Scenario 4 . . . 107
D.1.4 Scenario 5 . . . 107
D.2 Multiple inheritance involving an *implementation* . . . 108
D.2.1 Scenario 6 . . . 108
D.2.2 Scenario 7 . . . 108
D.2.3 Scenario 8 . . . 109
D.2.4 Scenario 9 . . . 109
D.2.5 Scenario 10 . . . 109
D.3 Multiple inheritance paths from the same root . . . 110
D.3.1 Scenario 11 . . . 110
D.3.2 Scenario 12 . . . 110
D.3.3 Scenario 13 . . . 111
D.4 Diamond-shaped multiple inheritance paths . . . 112
D.4.1 Scenario 14 . . . 112
D.4.2 Scenario 15 . . . 112
D.4.3 Scenario 16 . . . 113
D.4.4 Scenario 17 . . . 113

E Refactoring the `java.io` package 115
E.1 Result of refactoring . . . 115
E.2 Source code . . . 116
E.2.1 Source.java . . . 116
E.2.2 InputCode.java . . . 116
E.2.3 Sink.java . . . 118
E.2.4 OutputCode.java . . . 118
E.2.5 RandomAccessFile.java . . . 119
E.2.6 DataOutputStream.java . . . 120
E.2.7 DataInputStream.java . . . 121

List of Figures

1.1 Possible solutions for dual matrix representation problem 5

2.1 Java class (partial) hierarchies for I/O classes 8
2.2 Refactored Java class (partial) hierarchies for I/O classes 9
2.3 Class (partial) hierarchies for matrix/vector classes 10
2.4 Class (partial) hierarchies for matrix/vector classes with separation 10

3.1 Method visibility 21
3.2 Scenarios of inheritance conflict 23
3.3 Inheritance suspension scenarios 24
3.4 Dynamic semantics of multisuper call 27

4.1 Modified Java rules for a *type* declaration 32
4.2 Modified Java rules for method invocation 33
4.3 Schematic layout of an `ExpandedMethodTable` (EMT) 35
4.4 Multiple code inheritance scenarios 1 - 17 38

5.1 Run-time memory layout for *class*, *interface*, *implementation* and *object* data 43
5.2 Virtual method table (VMT) entries 48
5.3 Interface method table (IMT) entries 51
5.4 Excerpt of the run-time `ConstantPool` (RCP) for `DataInputStream` 56
5.5 Selected multiple code inheritance scenarios from Figure 4.4 62

6.1 Example for static method promotion 69
6.2 Example for prefix method promotion 71
6.3 Example for super-suffix method promotion 72
6.4 Hierarchy of the *classes* for data input/output before refactoring 73
6.5 Modified declarations for data input/output *types* in the `java.io` package 75
6.6 Hierarchy of the *classes* for data input/output 76

B.1 Schematic Diagram of the Modified Class File Structure 94

D.1 Legend for scenario diagrams 105
D.2 Scenarios 1 - 2 106
D.3 Scenarios 3 - 5 107
D.4 Scenarios 6 - 8 108
D.5 Scenarios 9 - 10 109
D.6 Scenarios 11a and 11b 110
D.7 Scenarios 12a and 12b 111
D.8 Scenarios 13a and 13b 111

D.9 Scenarios 14 and 15 112
D.10 Scenarios 16 and 17 113

List of Tables

4.1 Summary of changes (compiler portion emphasized) 29
4.2 Access flags interpretation by the Java VM 31
4.3 Compilation results for scenarios 1 - 17 39

5.1 Summary of changes (VM portion emphasized) 41
5.2 Conversion from execution bytecode to quick-ed bytecodes 58
5.3 Results of EMB lookup 63

6.1 Common code grouping by promotion technique for data input methods in `DataInputStream` and `RandomAccessFile` 74
6.2 Common code grouping by promotion technique for data output methods in `DataOutputStream` and `RandomAccessFile` 74
6.3 Change in the number of methods as a result of refactoring 76
6.4 Change in the number of lines of code as a result of refactoring 77
6.5 Compiler performance comparison 78
6.6 VM performance comparison 79

A.1 Meaning of the abbreviations used 91

E.1 Change in the number of executable statements as a result of refactoring . . 115

Chapter 1

Introduction

1.1 Overview

Since the development of Simula-67 by Nygaard and Dahl [27], object-oriented programming (referred to as OOP, hereafter) has become a popular programming paradigm. In the past three decades, many researchers have dedicated their efforts to improving this paradigm, and many programming languages supporting OOP have been developed. Some of the widely used commercial products include Java, Smalltalk, C++ and C#.[1]

The object-oriented programming paradigm is an extension of the concept of an Abstract Data Type (ADT) [25, 24]. A program developed using OOP can be viewed as a collection of reacting agents known as *objects*[2] interacting with each other through message passing.

An *object* is used to model a concept or a real-world entity. Every *object* has a *type*, an interface that defines the set of messages applicable to the *object*. A behavior describes the response (to be implemented in a method) of the *object* to a received message. A *class* is the programming unit responsible for creating object instances for a *type* and for mapping messages to behaviors.

In OOP, a problem is solved by interactive message-passing among *objects* that have private states [10]. This contrasts with the procedural programming paradigm, in which a problem is solved by changing the states of named storage locations according to a set of predefined instructions. OOP allows software to model the real world more closely, especially for complex systems in which there is a high degree of interaction among components of the system. It is a programming methodology that encourages modular design and code re-use. Many believe that programs developed using OOP techniques tend to be

[1] Trademarks of their respective owners.

[2] Section 1.4 describes the notation used in this dissertation.

more reliable, extensible and robust [22].

There are three major features that contribute to the success of OOP:

Encapsulation refers to the ability to protect the internal representation of an *object* from external access.

Inheritance refers to the ability of a *class* (*type*) to re-use properties of another *class* (*type*).

Polymorphism refers to the ability to map the same message to different methods in different *classes* at run-time.

Unfortunately, there are no standard accepted meanings of the notions of *type* and *class* in the OOP community. This dissertation adopts the ones that support a clean separation between the concepts of interface, implementation and representation (the internal data layout of an object) [21]:

1. An *interface-type* defines the set of legal operations (programmatic interface) for a group of *objects* that models a real-world concept.

2. A *code-type* implements an *interface-type* by associating code with methods that implement each of the legal operations defined by this *interface-type*, without constraining the data layout.

3. A *data-type* describes the data structure of the objects for the *interface-type*. It also provides the methods for accessing these data and a mechanism for creating *object* instances that conform to this *interface-type*.

The generic term *type* refers to any one of these three notions. Almost all OOP languages handle equivalence of *types* using the convention of *name equivalence*. Under this arrangement, two *types* with the same set of behaviors will not be treated as equivalent unless they also have the same name. *Subtyping* is a special relationship between two *types*, where the *subtype* guarantees behaviors conforming to those of the *supertype*. This can be viewed as an *is-a* relationship of *types*. Given that *type* T is a *subtype* of *type* S, an *object* of *type* T is implicitly of *type* S by *the principle of subsumption*.

A *class* is a programming language construct that implements the concept of *type* and is responsible for creating object instances for the *type*. Chambers [12] states that "If a *class* conforms to a *type*, then all of its instances support the interface specified by the *type*."

Parallel to the *subtyping* relationship between two *types* is the *subclassing* relationship between two *classes*. This refers to a strategy by which a *subclass* re-uses some or all of the implementations (code) from its *superclass*.

Since *type* refers to any one of the three notions mentioned above, inheritance, which refers to the ability of a *type* to re-use properties of another *type*, also has three distinct types:

1. *Interface inheritance* refers to the re-use of the operations of an *interface-type* by its *subtypes*.

2. *Code inheritance* refers to the re-use of the code in the parent *code-type* bound to an operation by its *subtypes*.

3. *Data inheritance* refers to the re-use of the object layout of the parent *data-type* by its *subtypes*.

Popular languages such as Java, C# and Eiffel combine *code-type* and *data-type* into one single construct, the *implementation-type*. *Implementation inheritance* refers to a combined *code* and *data inheritance*. Other popular languages, such as C++ and Smalltalk, further combine this *implementation-type* with *interface-type* into a single (*class*) concept.

With respect to the number of permissible direct *super-types*, there are two distinct kinds of inheritance:

1. ***Single inheritance*** allows a *type* to have no more than one direct *supertype*.

2. ***Multiple inheritance*** allows a *type* to have more than one direct *supertype*.

While C++ supports both multiple data inheritance and multiple code inheritance, C#, Smalltalk and Java support only single implementation inheritance. This dissertation describes a novel way to support the separation between *interface-type*, *code-type* and *data-type* in the Java Programming Language, and adds supports for multiple code inheritance.

1.2 Motivations

OOP permits application programs to model the real world more closely than programs developed using the procedural programming paradigm. One of the major advantages of OOP is the strong support of code re-use, which is done via the inheritance mechanism.

Ironically, the current form of inheritance imposes a restriction on code re-use in the available language systems. It is impossible to use the same implementation in two unrelated inheritance hierarchies without repeating the code.

The reason for this restriction is best explained by the fact that there is not a clear one-to-one correspondence between language features and concepts of *interface-type*, *code-type* and *data-type*. The code re-use problem can be solved more elegantly using features of a language that supports a clean and complete separation between these language concepts; a language of this kind would enable a program to use less code.

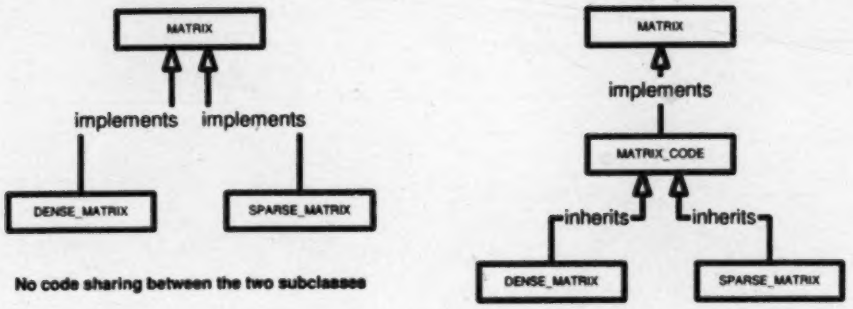
The Java Programming Language (referred to as Java, hereinafter) is one of the most widely used programming languages. It has rudimentary support for separate language constructs for *interface-type*, *code-type* and *data-type*, which are called *interface*, *abstract class* and *class*, respectively. Java supports multiple inheritance from *interfaces* but only single inheritance for *class*, whether the *class* is abstract or not.

From the software engineering perspective of code maintenance, it is best to have the minimum dependence of code on the actual data layout. Without independence, efforts to improve performance by optimizing data layout would require extensive changes to the codes that access this data directly.

For example, suppose a matrix is modeled using a `MATRIX` class. Traditionally, a matrix is represented using a block of memory units to store the values of each of the elements of the matrix. This block of memory can be visualized as a rectangular two-dimensional array. Sometimes, a different, more space-efficient, representation for a matrix is desirable, in which case a `SPARSE_MATRIX` class is required. One possible way to represent a sparse matrix is to use a doubly-linked orthogonal list structure storing only those non-zero members with their row and column numbers. Despite their different internal representations, the two classes should behave the same.

One current solution to this dual matrix representation problem is to have a `MATRIX` interface used as the *interface-type* for describing the behaviors of a matrix. The two representations of a matrix are implemented using two implementations of the `MATRIX` interface, `DENSE_MATRIX` and `SPARSE_MATRIX`. These two implementations are used as the *implementation-types* (a combined *code-type* and *data-type*) for the respective matrix representations. Figure 1.1(a) shows the schematic diagram of the relationship among the classes. Unfortunately, the two subclasses cannot share code using this solution, since interfaces in Java cannot contain code.

A solution to this problem is to add an abstract class `MATRIX_CODE` as the *code-type*



(a) Current solution

(b) Better solution

Figure 1.1: Possible solutions for dual matrix representation problem

for implementing the behaviors defined in the MATRIX interface. The two *subclasses* of MATRIX_CODE, DENSE_MATRIX and SPARSE_MATRIX, are used as *data-types* for implementing the data layouts and providing accessor functions to its internal data. The schematic diagram showing the relationship among these classes is shown in Figure 1.1(b). One typical example would be implementing the multiplication operation. This operation is implemented in MATRIX_CODE as a method that applies the dot product method to rows and columns. The dot product method itself is implemented in each of the two *data-types* – DENSE_MATRIX and SPARSE_MATRIX – as it requires access to the internal data of each representation.

Using an *abstract class* as a *code-type* in Java still restricts code re-use, as Java only supports single inheritance for *classes*. This dissertation proposes a solution to remedy this situation, by adding a new language construct at the source code level. The new construct, *implementation*, is a new form of *code-type* that supports multiple inheritance. It supports better code re-use through multiple inheritance.

1.3 Contributions

The work done in this thesis makes the following contributions:

1. It provides the first implementation using a virtual method table for an *interface* to extend Java to support the separation between *interface-type*, *code-type* and *data-type*.

2. It provides an implementation to extend Java to support multiple code inheritance

with changes at load time only.

3. It provides the first compiler support to the extended Java that supports multiple code inheritance and the separation between *interface-type*, *code-type* and *data-type*.

The language extension to Java allows elegant program architecture design and implementation to better model complex problem domains using Java.

1.4 Notation

Throughout this dissertation, the following scheme is used:

symbol	notation
term	a term in its first appearance with definition
class	a term used in the Object-Oriented Programming context
`keyword`	a keyword used in the Java grammar
`method()`	code related
invokevirtual	an opcode mnemonics used in a Java virtual machine instruction
`classclass`	an "in-memory" data structure
§	a section where a definition or a description can be found

1.5 Organization

In Chapter 2, several modeling problems encountered in Java are studied, and the weakness of Java in handling these cases is revealed. Extensions to the language to increase its expressive power are proposed in Chapter 3. Modifications made to the IBM Jikes Compiler and the Java Virtual Machine to support the proposed language extensions are presented in Chapters 4 and 5, respectively. The performance of the extended language system is evaluated in Chapter 6. Summary and conclusions are included in Chapter 7. A list of the abbreviations used in this thesis can be found in Appendix A. Appendix B shows the schematic layout of a `.class` file. A grammar for MCI-Java is included in Appendix C. Appendix D describes 17 multiple code inheritance scenarios which are commonly found in normal application programming. The result of refactoring the `java.io` package is detailed in Appendix E.

Chapter 2

Code Inheritance

Some problems in OOP cannot be solved without a separate *code-type*, and repeated code is unavoidable without multiple code inheritance. These problems are discussed in this chapter, with special reference to Java. A brief overview of multiple inheritance (§1.1) in other common OOP languages is included at the end of the chapter.

2.1 Common code

It is very common to find similar behavior exhibited by different species in the real world. Therefore, it would not be surprising to find different *classes* with common behavior. Many methods implementing the same behavior are expected to have code segments that are similar, if not exactly the same.

To avoid repeated code and facilitate easy maintenance, the common code segment should be kept in a common ancestor of the two relevant *classes* so that the common code is accessible by both of them. In Java, an *interface* cannot contain code; the common ancestor must therefore be a *class*. Since it is not desirable to have this ancestor restricting the data layout of the two *subclasses*, it is best to have an abstract *class* to act as the common ancestor. Each *subclass* can then have a data layout different from that of the other, which seems to provide a valid solution.

This solution does not work, however, when a *class* shares common behavior with more than one other *class* having different sets of behavior. The primary cause is due to Java only supporting single inheritance (§1.1) in *class*, in contrast with multiple inheritance in *interface*. The *class* needs to inherit from more than one abstract *class* in order to be able to use the shared code for all the common methods. This violates the rule for single inheritance in *classes*.

A typical example of the problem can be found in the `java.io` package. Two *classes*,

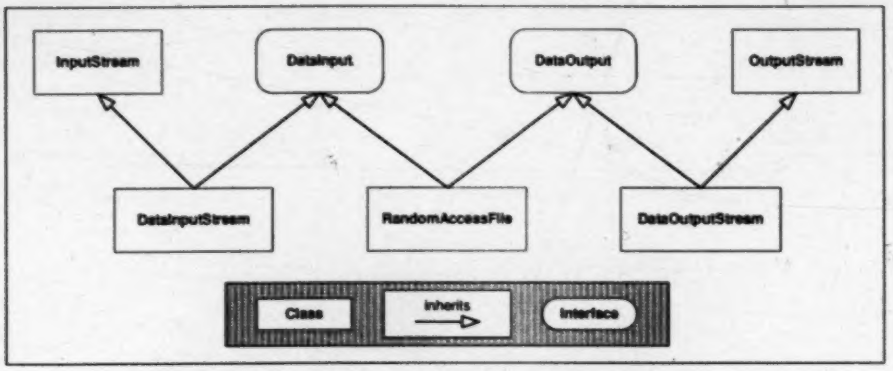


Figure 2.1: Java class (partial) hierarchies for I/O classes

`RandomAccessFile` and `DataInputStream`, have common behavior specified by the *interface* `DataInput`. Similarly, `DataOuput` describes the behavior applicable to both `RandomAccessFile` and `DataOutputStream`. The relationship among the *classes* is depicted in Figure 2.1.

While it is possible that `RandomAccessFile` and `DataInputStream` share code stored in one abstract *class*, and `RandomAccessFile` and `DataOutputStream` use common code kept in another abstract *class*, single inheritance means that the single *class* `RandomAccessFile` cannot inherit from both abstract *classes*.

To solve this problem, it is necessary to have a *type* declaration from which a *class* is allowed to inherit multiply. The *type* declaration should be allowed to contain code only, not data layout. The suggested refactored *type* hierarchy is depicted in Figure 2.2.

In Java, a *class* is allowed to inherit from multiple *interfaces* but an *interface* is not allowed to contain code. An abstract *class* can contain only code without data layout. However, only *single inheritance* is supported in *classes*. A new *type* declaring unit may therefore be required to solve the problem.

2.2 Multiple representation

In Section 1.2, the benefit of having dual representations for a matrix was discussed. Optimization can be obtained by using data representations optimized for special scenarios. The advantage of having multiple representations for a single *type* is explained in a quote excerpted from an online tutorial[1] [1] offered by Sun Microsystems Inc. `List` is an

[1]The tutorial is on "General Purpose Implementations" using the Collection framework.

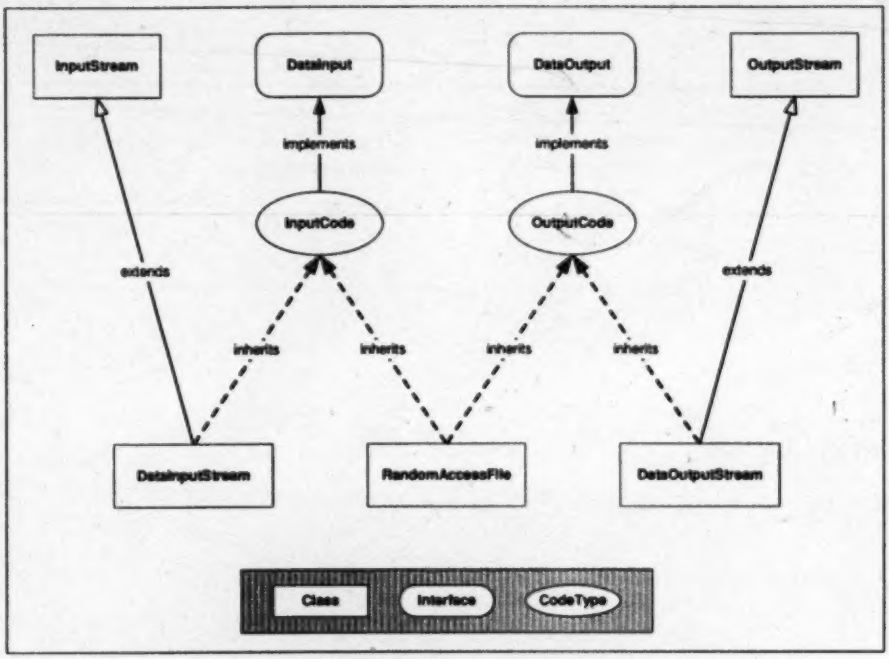


Figure 2.2: Refactored Java class (partial) hierarchies for I/O classes

interface with two *classes* that implement it, `ArrayList` and `LinkedList`.

> The two general purpose List implementations are ArrayList and LinkedList. Most of the time, you'll probably use ArrayList. It offers constant time positional access, and it's just plain fast, because it does not have to allocate a node object for each element in the List, and it can take advantage of the native method System.arraycopy when it has to move multiple elements at once. Think of ArrayList as Vector without the synchronization overhead.
>
> If you frequently add elements to the beginning of the List, or iterate over the List deleting elements from its interior, you might want to consider LinkedList. These operations are constant time in a LinkedList but linear time in an ArrayList. But you pay a big price! Positional access is linear time in a LinkedList and constant time in an ArrayList. Furthermore, the constant factor for LinkedList is much worse. If you think that you want to use a LinkedList, measure the performance with both LinkedList and ArrayList. You may be surprised.

Without a separate *code-type*, the problem discussed in Section 1.2 is still solvable by implementing methods specified in the `MATRIX` interface in `MATRIX_CODE`, an abstract *class*, extended by the two *classes*, `DENSE_MATRIX` and `SPARSE_MATRIX`, which are the *data-types* for the two distinct representations of a matrix. The class hierarchy is shown in Figure 1.1(b). However, in the scenario that follows, the problem cannot be solved with the present language structure of Java.

A column or row vector is a specialized form of a matrix. Some of the matrix operations can be optimized for this special form. Thus, some methods which are defined for matrix

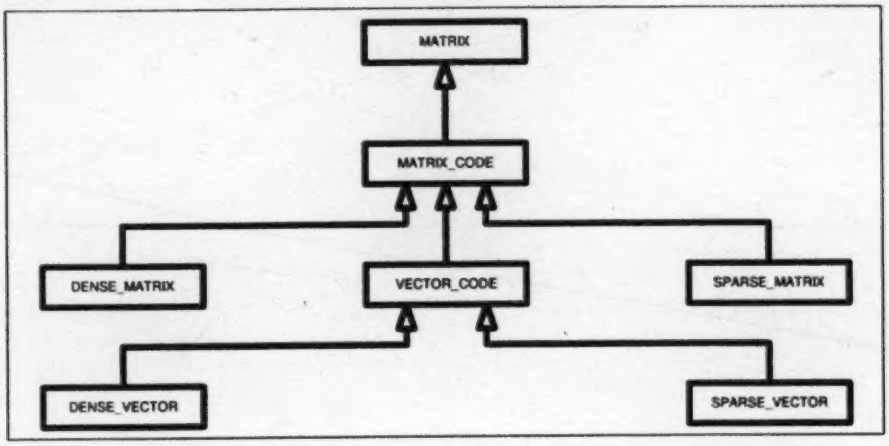


Figure 2.3: Class (partial) hierarchies for matrix/vector classes

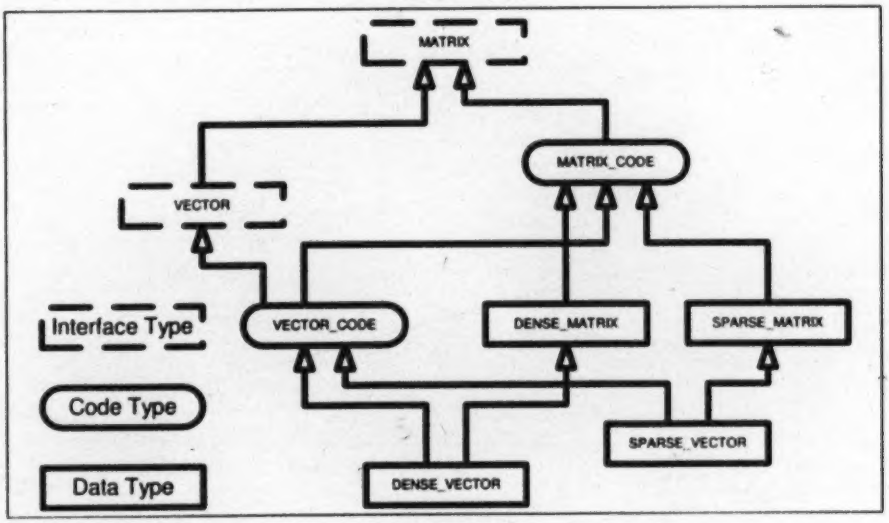


Figure 2.4: Class (partial) hierarchies for matrix/vector classes with separation

operations can be implemented differently, optimizing for column or row vectors. However, these implementations are good only for vectors, but not matrices. Furthermore, it is very reasonable to have dual representations for vectors, making each representation of vector a specialization of its counterpart for matrix.

In Java, one possible way to model the matrix/vector scenario is depicted in Figure 2.3. The class hierarchy for matrix, as shown in Figure 1.1(b), remains unchanged. The abstract *class* `MATRIX_CODE` is extended by `VECTOR_CODE`, an abstract *class* containing

code specialized for vectors. The two *data-types* representing the dual representations of a vector become *subclasses* of `VECTOR_CODE`. This model is very close to the real situation capturing almost all relationships among the *classes*, except those among the *data-types*. `DENSE_VECTOR` should be a specialization of `DENSE_MATRIX`, while `SPARSE_VECTOR` should be a specialization of `SPARSE_MATRIX`.

With separate language constructs for *interface-type*, *code-type* and *data-type*, the matrix/vector case can be modeled using a class hierarchy, as shown in Figure 2.4. It can be seen that all specialization relations are reflected in the class hierarchy. Data layouts of the matrix *classes* are re-used in the vector *classes*. This factorization is not possible in Java, since `DENSE_VECTOR` and `SPARSE_VECTOR` must both use multiple code inheritance.

It is obvious that programmers are able to model real world problems more closely using languages that support the separation between *interface-type*, *code-type* and *data-type*. Java does not support the separation between these *types*, and this dissertation proposes a modification to the language to move it in this direction.

2.3 Overview of OOP languages

Several currently available static-typed OOP languages are studied according to their published specifications to determine how they are designed in terms of inheritance and separation. The languages studied include BeCecil [13], Cecil [11], C++ [32], C# [5], Eiffel [26], Java [18] and Sather [31].

2.3.1 Separation

Not all languages studied provide support for separation between *interface-type*, *code-type* and *data-type*.

C++ gives the modular unit dual functional roles. A class is used for declaring an *interface-type* for programmatic interface, *type* implementation and instance creation.

Eiffel uses a class to declare a *type* and to create object instances of the *type*. There is no separation between *type* and *class*, although deferred classes permit a declared feature to be implemented later by another class inheriting from it.

Both Java and C# provide moderate support for the separation between *interface* and *implementation* through `interface`, a modular unit in addition to `class`. It allows the definition of an *interface-type* without requiring implementation to be included. The presence of `interface` allows a `class` to have additional sets of behavior on top of the one inherited from the *superclass*.

Sather provides a mechanism that supports separation between *interface* and *implementation* in that classes are used to define code and storage. With the presence of this mechanism, class inheritance is split into two parts. One part of the inheritance deals with *type* relationships between classes (subtyping) and the other part (code inclusion) deals with code relationships between classes.

Cecil and BeCecil (an extension of Cecil) support a polymorphic static type system which distinguishes between subtyping and code inheritance. The static type system makes *type* declaration and *subtyping* independent of objects and their creation. Cecil and BeCecil do not provide mechanisms to separate code from representation; the two are blended into an *implementation-type*. Code inheritance must be accompanied by preservation of data layout.

2.3.2 Code inheritance

Both Java and C# allow only single inheritance in *classes*. Multiple inheritance is supported through using an interface which contains no code. In other words, these two languages support multiple inheritance in *interface-type*, but not *code-type*. As no code is allowed in *interface*, one would expect a considerable amount of duplicated code when implementing multiple inheritance using these two languages.

C++ supports multiple inheritance in both *interface-type* and *code-type*. However, there is no separation between them; inheritance in one is implicitly imposed with inheritance in the other. When C++ encounters a diamond-shaped inheritance (inheriting from two modular units having a common source of inheritance), an explicit qualifier must be created to refer to the common root.

Eiffel also supports multiple inheritance in both *interface-type* and *code-type*. Under its terminology, multiple inheritance refers to the V-shaped inheritance (inheriting from two modular units having no common source of inheritance) while the diamond-shaped inheritance is referred to as repeated inheritance. The repeated common root is treated differently, as in C++. The programmer has the option of sharing the common root and making it one single entity, or replicating the common root through renaming.

Sather allows multiple *type* inheritance. Since *subtyping* is only possible with abstract types, no diamond-shaped *type* inheritance is possible. When it comes to diamond-shaped code inheritance, Sather adopts the same scheme used in C++, requiring an explicit qualifier for methods coming from the common root.

Cecil permits multiple inheritance in both *type* and code. It uses partial ordering to

handle diamond-shaped inheritance. An implementation is regarded as more specific if it is located closer to the class where the call is made. The most specific one is to be invoked. BeCecil is a language that supports multi-dispatch and thus handles code inheritance differently.

Neither C++ nor Java permits methods defined in another *class*, which is not an ancestor of the current *class*, to be inherited.

Sather provides code inheritance through *subclassing*. Each time the keyword `include` is used, the implementation of a specified *type* is incorporated. This is different from inheriting code without an association of a *type*. Sather's special feature – `closure` – is equivalent to a function pointer in the other languages.

Although Eiffel supports multiple and repeated inheritance, there is no mechanism to inherit code without any *type* association. A programmer may elect to inherit from a *type* while rejecting some of the methods declared in the *type*. However, the reverse is not possible. The programmer cannot accept a method while rejecting its containing *type*.

2.4 Summary

OOP is intended to promote code re-use. Ironically, rather than providing the necessary support to it, most of the currently available languages restrict or do not allow code re-use.

When using languages that do not support the total separation between *interface-type*, *code-type* and *data-type* and which only support single inheritance in *class*, it is not possible to implement, without duplicated code, a *class* which shares common behavior with more than one *class*, each of which has different behavior sets.

Optimization can be obtained through different data layouts, each specialized for a unique scenario. However, multiple representations of an interface cannot be implemented in a way that models the real situation closely without total separation between *interface-type*, *code-type* and *data-type*.

No single commonly used OOP language provides a solution to these modeling problems. A proposal to extend Java, (which is a commonly used OOP language), to support multiple code inheritance and the separation between *interface-type*, *code-type* and *data-type* without changing the original semantics, will be discussed in Chapter 3.

This page contains no text

Chapter 3

Extensions to the Java Programming Language

Several weaknesses of Java, in terms of expressiveness, have been identified in the previous chapter. These include difficulties in modeling a real problem, with respect to multiple representations, multiple behavior and shared code. This chapter describes MCI-Java, an extension to Java, which is proposed as a solution to the problem. The chapter begins with the establishment of a set of guidelines to be used as criteria for the design decision. This is followed by a section on the various orthogonal dimensions in OOP language design, on which the proposed solution is based. While the actual implementation is detailed in the following two chapters (Chapters 4 and 5), design-related issues and the semantics of the added features are discussed in Section 3.3.

3.1 Guidelines for changes

One of the strengths of Java is its platform independence. The component responsible for this machine- and operating system-independence is the Java Virtual Machine (JVM), which is an abstract computing machine with its own instruction set (the bytecodes). Compiled code is represented in a binary form known as the `.class` file format, which is machine- and operating system-independent. A `.class` file contains the bytecodes, a symbol table (the constant pool) and other ancillary information. There is a strong format and structural constraint imposed by the JVM on the `.class` file format.

The Java Virtual Machine Specification (JVM Specification) [23] does not assume any implementation detail. However, any JVM implementation must be able to read the `.class` file format and correctly perform the operations specified there, according to the JVM Specification. While the implementation details are not precisely specified by the

JVM Specification, the run-time environment should have:

1. various run-time data areas to be used during program execution, as well as lookup and symbolic resolution routines that access them,

2. a class loading mechanism which can translate symbolic references contained in a `.class` file into concrete method references within the run-time environment, and

3. a bytecode interpreter for executing the bytecodes correctly.

The interpreter is machine-specific and is usually highly optimized for performance. None of the changes made to the bytecode execution procedure can be made in the interpretation section of the JVM, to avoid affecting performance. Modifying the highly optimized bytecode interpreter would be a maintenance disaster, given the numerous machine specifications on the market. Therefore, all changes to the JVM must be in the class loader or symbolic resolution code.

Very few OOP languages possess the modularity features provided by Java. By using this feature, a single compilation unit – usually a top level *type* (a *class* or an *interface*) – can be recompiled without requiring other units using this *type* to be recompiled; and an application using this recompiled *type* can still be executed without recompiling the application. Other common languages such as C++ and Smalltalk require a recompilation of all related files, once changes are made to one of the files, before the affected application can be executed. MCI-Java is designed to behave like Java, with no recompilation required.

Another important consideration is backward compatibility. The availability of a standard run-time library (the file `rt.jar`), which contains an archived collection of the built-in classes, is another important feature of Java. All programs written in Java use these classes both implicitly and explicitly. All the changes made to these classes must be transparent so that the legacy codes already in existence are not affected, if the code can still be executed in classic JVM. Because of the existence of numerous applications written in Java, the semantics of the original language should not be changed.

In view of the above, it was decided that MCI-Java should meet the following criteria:

1. No changes should be made to the highly optimized bytecode interpreter and thus the bytecode execution procedure of classic Java should not be modified.

2. Modularity should be supported and the original semantics of the existing features, as specified by the Java Language Specification [18], should not be modified nor affected by any new features introduced in MCI-Java.

3. Modifications to the `.class` file and the `rt.jar` file , if any, should conform to the current specifications described in the JVM Specification and the Java Application's Programming Interface (API) [2], respectively.

3.2 Orthogonality in language design

Wegner has identified six orthogonal dimensions of OOP language design. "These dimensions provide a design framework for object-oriented languages in terms of computing agents, classification mechanisms, sharing mechanisms, and interface specification mechanisms." [33]

The six dimensions are orthogonal in the sense that no single one is the consequence of any of the others. They are named below, four of them being accompanied by the definitions provided by Wegner:

1. **Object:** An object has a set of "operations" and a "state" that remembers the effect of operations.

2. **Type:** A type is a behavior specification that may be used to generate instances having the behavior.

3. **Delegation:** Delegation is a mechanism that allows objects to delegate responsibility for performing an operation, or finding a value, to one or more designated "ancestors".

4. **Abstraction:** A data abstraction is an object whose state is accessible only through its operations.

5. **Concurrency**

6. **Persistence**

As concurrency and persistence are outside the scope of this thesis, they will not be discussed here.

Wegner's definitions of "Object" and "Type" are similar to the definitions of *object* and *type* found in Section 1.1 of this thesis. Inheritance, one of the three main features of OOP, can be viewed as a specialization of "Delegation", while encapsulation is an application of "Abstraction".

Leontiev, Özsu and Szafron [21] have suggested the total separation of interface, implementation and representation for objects in object-based database management programming languages. Different and unique language constructs are required for each of the three concepts. The idea of total separation is parallel to Wegner's concept of orthogonal design dimensions, with slight modifications.

Since an *interface-type* defines the set of legal operations for a group of *objects*, it can be seen as a feature in the "Type" dimension. These *objects* delegate the responsibility for performing an operation to a *code-type*, which implements the methods defined by an *interface-type*. Accordingly, a *code-type* belongs to the "Delegation" dimension. A *data-type* contains the data layout and a set of accessors for accessing the internal state of the *object* it represents. If Wegner's concept is slightly modified so that an "Object" has a state and a set of accessors (instead of "operations"), the *data-type* defined in [21] will fit perfectly into the "Object" dimension. The concepts of separation and orthogonality both emphasize the independence between any two of the language features mentioned in their proposals.

The orthogonal design dimensions identified by Wegner are useful tools to understand issues about language design. They provide a platform to search for solutions. The total separation of *interface-type*, *code-type* and *data-type* suggested by Leontiev *et el.* is a realization of this concept of orthogonal dimensions, with each of the three *types* belonging to a different orthogonal dimension.

A language can be more expressive if all of the features are so designed that they belong to a unique dimension. Using orthogonality, any operation in one of the features will have no effect on another feature. MCI-Java, the proposed extension to Java, conforms to this framework.

3.3 MCI-Java

3.3.1 Implementation

MCI-Java introduces *implementation* as the third *type*-declaring unit at the source code level, in addition to the two currently used in Java. A distinct language construct is therefore used for each of the three totally separated notions of *type* suggested by Leontiev *et el.* [21]. At the Virtual Machine (VM) level, an *implementation* is compiled to an *interface* containing code. Therefore, only the two existing mechanisms – one for *interface* and the other for *class* – are used, without the need to introduce a new one.

This two-level setup decouples language syntax changes from the JVM support required for code in *interface* and multiple code inheritance. There can be different language constructs and syntax at the source code level with different compiler implementations to support them. The modified VM will be able to execute bytecode generated by compilers which support code in *interface*. For example, if someone decides to extend Java to allow code in *interface* without introducing a third *type*-declaring unit, the modified VM will still support code generated from the modified compiler.

At the source code level, the original version of Java has two *type* declaring units - *class* and *interface*. An *interface* in Java is used to specify the behavior of a group of *objects*, while a *class* is used to define the data layout of *objects* and to associate code with methods for implementing behavior. In other words, while an *interface* contains abstract methods (with no code), a *class* can have data for state and code for methods. This setup must be preserved according to the guidelines set out in Section 3.1.

It is a natural choice to assign the role of an *interface-type* to an *interface* because this is the role assigned to it in the original version of the language. In order for the original semantics of the language to be preserved, it is necessary that data layout can be defined in a *class*. This is obviously the role of a *data-type*.

An abstract *class* in Java seems to be a good choice for *code-type* because no instance of the class can be created. The discussion in Chapter 2 has clearly indicated that multiple code inheritance is a necessary part of the solution. However, if an abstract *class* is used as a *code-type*, the original semantics of the language have to be modified to support *multiple inheritance* in *classes*. This cannot be done according to the guidelines set out in the previous section. Therefore, a new construct is introduced to the language at the source code level for *code-type*.

At the VM level, the same "in memory" data structure is used for representing a *class* and an *interface*, in Sun's implementation of the JVM. Obviously, this data structure is capable of:

1. storing information for a *type*-declaring unit,

2. keeping method details, including code (concrete methods in *class*), and

3. supporting both single (for *class*) and multiple (for *interface*) inheritance.

An *implementation* is a *type*-declaring unit which contains code and supports *multiple inheritance*. The existing data structure already has all the features required by an *im-*

plementation and it was therefore decided to use the same "in memory" data structure to represent an *implementation* at the VM level.

3.3.2 MultiSuper method invocation

In classic Java, a super method invocation (the super call) is used to access an overridden instance method of the immediate superclass. Since MCI-Java supports multiple code inheritance, it is possible that a method declared in an *implementation* is overridden in the current *class*. A method invocation mechanism analogous to the super call is therefore required to access an overridden method declared in an *implementation*. Multisuper method invocation (the multisuper call) is an extension to the super call and is used to invoke an overridden method of an immediate *implementation* inherited by the current *class* to which the receiver object belongs.

3.4 The semantics of multiple code inheritance in MCI-Java

3.4.1 Inheritance and visibility

According to the original semantics of Java, a method declared in a *class* higher up in the inheritance hierarchy is available for use by instances of the current *class*, by virtue of inheritance. It is said to be visible in the current *class*. However, if another method with the same signature, formed with the method name and the number and types of formal parameters to the method, is declared in the current *class*, the aforementioned "inherited" method will be overridden, and will not be visible in the current *class* and its subclasses.

The idea of visibility is extended to include *implementation*. Code visible in an *implementation* will be the one that is explicitly declared in this *type*, or inherited by this *type* from another *implementation*, and not overridden. While a *class* can inherit from an *implementation*, the reverse is not allowed. Both *class* and *implementation* can inherit from more than one *implementation*. A method in an *implementation* is visible in a *class* which inherits from this *implementation*, if no method with the same signature is defined in this *class*.

When a method is visible in a *class* or an *implementation*, the semantics are consistent regardless of whether the method is explicitly declared in this *type* or inherited from another *type*. In the example depicted in Figure 3.1 where the source code has a form similar to the one shown in Figure 3.1(a), method `m()` is declared in *implementation* `A`. Figure 3.1(b), shows that a *class* is denoted using a rectangle while an *implementation* will be represented

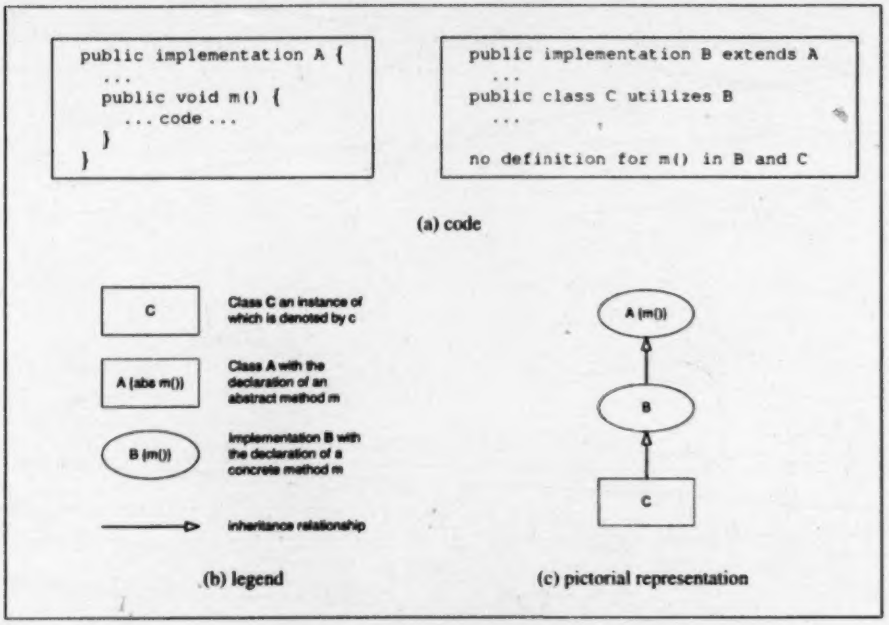


Figure 3.1: Method visibility

by an oval shape. If the alphabet denoting the name of the *type* is followed by a method name enclosed within a pair of curly brackets, it indicates that a definition of the method is visible in this *type*. The definition may be an inherited definition, or one that is explicitly defined in this *type*. This method is visible in *implementation* `B` as well as in *class* `C` by virtue of inheritance, as shown in Figure 3.1(c). If the method `m()` is declared in another *implementation* from which `A` inherits, and is visible in `A`, the situation will be the same as depicted in this diagram.

3.4.2 Precedence of supertypes

The original semantics of Java does not put any weight on the lexical order of a *supertype* within the declaration statement, in contrast to some languages with multiple code inheritance, such as CLOS [8], in which the lexical order is significant. For Java, a change in the order of declaration will not have any effect in any aspect; all methods have the same degree of importance. In order to preserve this situation, as required by the guidelines set out in Section 3.1, no precedence will be given to a particular *implementation* by virtue of its lexical order. For example, the method `m()` declared in `A` and the one declared in `B` will have equal precedence with the following two declaration statements for `C`:

1. `public class C utilizes A, B`

2. `public class C utilizes B, A`

Furthermore, an *implementation* is given the same status as a *class* with reference to being a source of code. As will be discussed in the following section, when resolving a conflict, no precedence will be given to a *class* or an *implementation* for its code. In other words, the method `m()` declared in class `C` will not be given any preference simply because it is declared in a class over another method `m()`, declared in an *implementation*.

3.4.3 Inheritance conflict and resolution

Huchard [19] has identified two kinds of conflict arising from code inheritance:

1. A *name conflict* occurs when more than one semantically distinct method uses the same name. For example, an instance of an Integer class uses the method `add(int)` to compute a sum, while an instance of a List class uses the method `add(int)` to add an integer value to its content. The two methods have the same signature but different semantics.

2. A *value conflict* arises when one method has more than one definition visible at the same time. An array-based list and a linked list will have different implementations for the same method `add(int)`.

The inheritance mechanism will try to resolve a *value conflict* but is not able to deal with a *name conflict*. Therefore, *name conflict* will not be discussed in this thesis. If a conflict cannot be resolved, an error of ambiguity will be raised either at compile time or at run-time, depending on when the conflict occurs. *Value conflict* is avoided in classic Java because of its overriding system and the fact that there is only single code inheritance. However, it cannot be avoided in MCI-Java because multiple code inheritance by a *class* via *implementation* is permitted.

There are two common strategies – linearization and graph-oriented – to resolve a conflict arising from multiple code inheritance. Snyder [30] has an overview on both.

The linearization strategies transform the partial ordering of classes into a total order. The final ordering is based on the relative order among classes within a list of direct superclasses. During the transformation process, unrelated classes may be inserted between a class and its parent. This ensures single inheritance, although there may be conflicting properties to be inherited. If it is necessary for a class to inherit those conflicting properties,

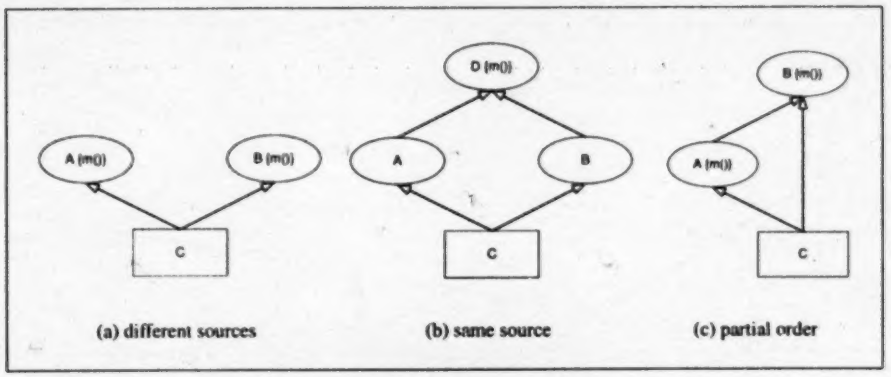


Figure 3.2: Scenarios of inheritance conflict

renaming is required. A typical example would be the "rename" subclause used in Eiffel [26] to avoid name clashes. In other words, code cannot be reused without transformation. This contradicts the modularity feature of OOP which ensures that each class can be designed and implemented independently of other unrelated classes. Furthermore, there is an additional major problem involving the ability of a class to communicate with its "real" parent.

Graph-oriented strategies resolve a conflict without requiring class transformation. When there is a conflict, the programmer will manually create a new definition, at the point where the conflict is observable, to override the conflicting definitions. Another way to avoid the conflict is to specify the source from which the property has to be inherited. This kind of strategy is used in C++ [32]. With this specification, the method lookup process, which usually starts from the receiver's class, will now start from the specified class. This hinders the capability of the run-time machine to locate the most suitable method within the inheritance hierarchy tree.

The strategy used in MCI-Java is a blend of the two. No class transformation is necessary to resolve a conflict, and a new definition at the point where the conflict is seen can remove the conflict by overriding it. When such a new definition is absent, a strategy similar to linearization is adopted. This strategy can be summarized as follows:

1. When the conflicting definitions actually come from the same source, as depicted in Figure 3.2(b), it is not regarded as a conflict.

2. In situations similar to the one shown in Figure 3.2(a), when there is no inheritance path linking the two *types* containing the conflicting definitions, it is ruled as un-

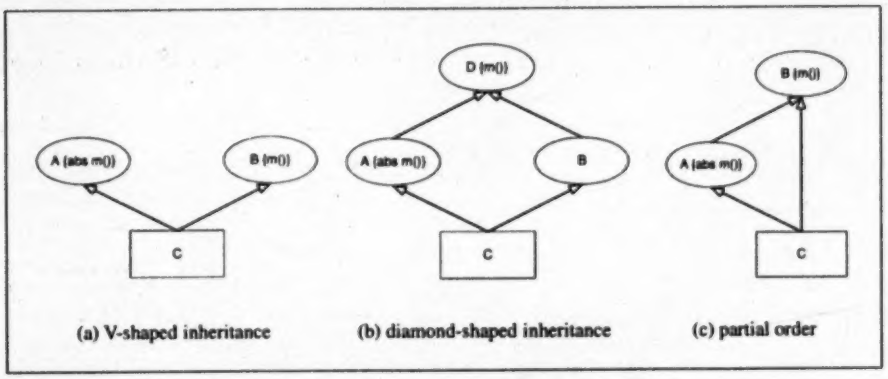


Figure 3.3: Inheritance suspension scenarios

resolvable and the compiler should react accordingly. At run-time, an ambiguity exception will be raised.

3. Figure 3.2(c) shows the scenario in which there is an inheritance path linking the two *types* containing the conflicting definitions. The method defined in the *type* which is lower in the inheritance hierarchy (partial ordering) takes precedence according to the *relaxed multiple code inheritance* [28], and thus no conflict will be reported.

3.4.4 Inheritance suspension

Section 8.4.3.1 of the *Java Language Specification* [18] states that "An instance method that is not abstract can be overridden by an abstract method." When this happens, there is an inheritance suspension.

Inheritance suspension is used to force a new implementation in the subclass chain. In other words, definitions found in the hierarchy tree above the *class* that creates the suspension are not intended to be used by its *subclasses*.

With inheritance suspension in mind, the scenario depicted in Figure 3.2(c) becomes a difficult-to-decide situation if `m()` in `A` is an abstract method. The new situation is now shown as Figure 3.3(c). The purpose of inheritance suspension is to block the reuse of inherited code (the one defined in `B` in this case). It is therefore against the intention of the original Java semantics to allow the code found in `B` to be reused in `C`. On the other hand, it can be argued that the suspension is intended to block code reuse along a path. The code from `B` is still visible in `C`, along another path. Thus, use of the code in C should be permitted.

Assuming that method `m()` is an abstract method in the situation described in Figure 3.2(a), the changed situation can be seen in Figure 3.3(a). The purpose of this abstract method declaration is to define a protocol, such that an implementation must be provided (for use) in `C`. The definition declared in `B`, visible in `C` would serve to fulfill this requirement. It is therefore natural to allow code in `B` to be reused in `C`, despite the fact that there is an abstract method declaration in `A`.

For consistency, the same resolution should be arrived at in situations depicted in Figures 3.3(b) and 3.3(c). To conclude, a concrete method takes precedence over an abstract method in conflict resolution.

3.4.5 Keywords `this` and `super`

No change is made to the semantics and use of the keyword `super` while the semantics of the keyword `this` is expanded in MCI-Java. Section 15.8.3 of the *Java Language Specification* [18] states that

> When used as a primary expression, the keyword `this` denotes a value, that is a reference to the object for which the instance method was invoked, or to the object being constructed.

The specification then explains (in Section 15.11.2) the use of the keyword `super` to access members of the superclass of a `class`, an instance of which is denoted by the keyword `this`.

According to the guidelines set out in Section 3.1, the semantics of the two keywords, `this` and `super`, must be preserved. However, the semantics of the keyword `this` are expanded to cover its usage in MCI-Java for the new feature - *implementation*.

As stated in the *Java Language Specification*, the keyword `this` refers to the receiver *object* of the sent message (from which the instance method is dispatched). Methods declared in an *implementation* are also instance methods. Thus, the use of the keyword `this` in instance methods declared in a *class* is extended to be applicable to methods declared in an *implementation*. When used in code in an *implementation*, `this` is not referring to an instance of the *implementation* but is used as a placeholder for the reference of the instance object that `utilizes` the code.

While the semantics of the keyword `super` have been preserved in MCI-Java, its use is limited to inside instance methods declared in a *class*. Using the keyword `super` in a method contained in an *implementation* is illegal since the compiler has no way to check if

all *classes* that `utilize` the *implementation* have implementations of that method in their superclasses.

3.4.6 Inheritance scenarios for MCI-Java

Many multiple code inheritance scenarios arise from real problem modeling (some very common ones are recorded in Appendix D for reference). For each scenario listed in Appendix D, there is a brief discussion on the results of applying the MCI-Java semantics of multiple code inheritance. The scenarios included in the discussion are not meant to be exhaustive, but are representative of the most common situations.

3.5 The semantics of multisuper call in MCI-Java

The multisuper call provides a mechanism to invoke an overridden method declared in an *implementation* from which the current *class* inherits either directly or indirectly. This is analogous to the super call used to invoke an overridden method declared in the chain of *classes*.

In C++ [32], each call to a non-virtual method declared in one of the *superclasses* is a statically compiled subroutine jump. This is different from the dynamic semantics of a Java method invocation which supports modularity (§3.1).

To make a multisuper call, the programmer is required to specify the name of the method, as well as the root of an inheritance path from which the method search starts. The search process is the same as that for an instance method in classic Java; the only difference between the two search processes lies in the search path. The search for an instance method in classic Java is done along the inheritance chain in which only *classes* are found. In the search for a multisuper instance method, the path starts from the specified *superimplementation* and travels up the inheritance path in which only *implementations* exist.

The scenario, as shown in Figure 3.4, is used to explain the dynamic semantics and the search process. Figure 3.4(a) shows the situation after the original compilation. The message `super(B).m()`, sent to an instance of `C`, triggers a search process starting from `B`, as specified. The definition in `D` is located. As a result, the message `super(B).m()`, sent to the instance of `C`, is dispatched to this definition.

A new definition for `m()` is then introduced to `B`. After the recompilation of `B`, the situation is depicted in Figure 3.4(b). Recall that the other *types* are not required to be recompiled, because of modularity. At run-time, the same message sent to the same instance of `C` triggers the same search process. However, this time, the new definition of `m()` in `B` is

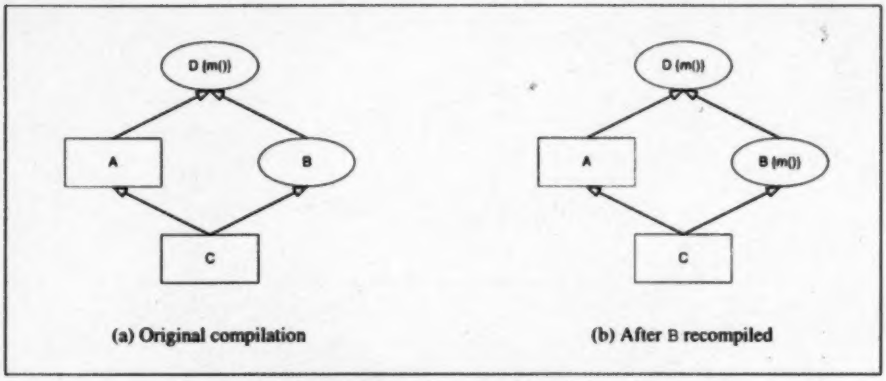


Figure 3.4: Dynamic semantics of multisuper call

located. The message `super(B).m()`, sent to the same instance of `C`, is now dispatched to this new definition, instead of to the definition found in `D`.

It can be seen from the example described above that the dynamic semantics of the multisuper call guarantees that the most specific method will be dispatched at run-time, despite the possible recompilation of the intermediate *types*.

3.6 Summary

MCI-Java, which supports a unique language construct for each of the three notions of a *type* and supports multiple code inheritance, is introduced in this chapter. The total separation of *interface-type*, *code-type* and *data-type* is able to remove some of the weaknesses in Java, that were identified in Chapter 2.

At the source code level, the two existing *type*-declaring units, namely *interface* and *class*, are used for declaring *interface-type* and *data-type* respectively. This necessitates the introduction of *implementation*, which is used for declaring a *code-type*. At the VM level, no new concept is introduced. The same "in memory" data structure is used for all three units and an *implementation* is compiled to an *interface* containing code. This allows the existing mechanisms in Sun's implementation of JVM to be used for the new feature.

While the original Java language semantics are preserved, *implementation* supports multiple code inheritance. The mechanism for resolving conflict arising from multiple code inheritance paths was discussed in detail, in Section 3.4.3.

Multisuper method invocation is introduced to permit invoking an overridden method declared in an *implementation*. A more dynamic semantics than the static semantics used

for a multi-super call in C++ is defined for the newly introduced multisuper call.

Modifications to Jikes to support MCI-Java are described in Chapter 4. The implementation details of the new features in Sun's JVM will be discussed in Chapter 5.

Chapter 4

Compiler Support

MCI-Java was introduced in Chapter 3. For each new language concept or feature, changes are made at both the compiler (source code) level and the VM level (to be discussed in Chapter 5). The changes are summarized in Table 4.1 for easy reference.

	Compiler	Virtual Machine
`.class` file	Introduce a new access control flag for a *class* to indicate an *interface* containing code (§4.1)	Modify code verification procedure during *class* loading to recognize an *interface* with code (§5.2)
type declaration	1. Introduce the new keyword `implementation` to indicate a declaration for *code-type* and the new keyword `utilizes` to indicate an inheritance from a *code-type* by a *class* (§4.2) 2. Modify the Java grammar to include the new keywords and introduce an option to accommodate for multisuper method invocation (§4.2)	No changes are required since an *implementation* is an *interface* with code at the VM level
inheritance and resolution	Introduce a modified algorithm for resolving method inheritance and conflict to identify the most specific method for invocation (§4.3)	Extend the use of the virtual method table (VMT) for both a *class* and an *implementation* (§5.3.3) and modify the method lookup routine for the bytecode *invokespecial* (§5.4.4)
method dispatch	Extend the semantics of *invokespecial* to include multisuper method invocation (§4.4)	No changes are made since all method invocations use the existing routines for dispatch

Table 4.1: Summary of changes (compiler portion emphasized)

Modifications made to the IBM Jikes Compiler [4] (hereinafter referred to as Jikes) to support the new proposed features will be discussed in this chapter. Jikes has been chosen as the compiler to be modified to support MCI-Java for the following reasons:

1. It is an Open Source Initiative (OSI) [6] certified software, making free distribution possible.

2. It is strictly Java-compatible, adhering "*to both The Java Language Specification and The Java Virtual Machine Specification as tightly as possible, and does not support subsets, supersets, or other variations of the language.*" [4]

Modifications to Jikes are made in the following areas and will be discussed in this chapter in the order presented:

1. the access control flags used in the `.class` file - the binary form of a module (usually a top level *type*),

2. the syntactic grammar for Java to accommodate the new concept of *implementation* and the new feature – multisuper method invocation,

3. the mechanism for handling code inheritance and dealing with conflict resolution, and

4. bytecode emission (for multisuper call).

4.1 The extended `.class` file

Each `.class` file (§3.1) contains the definition of a single top-level *type* which may be a *class*, an *interface* or an *implementation* (in MCI-Java only). A schematic diagram of the entire structure of a `.class` file can be found in Figure B.1 in Appendix B. A `.class` file uses variable-sized items to store attributes and instructions. These information items make references to the symbolic information stored in the constant pool, a component of the `.class` file. At run-time, the symbols stored in the constant pool will be transferred to the "in memory" run-time constant pool and resolved into actual linkage references as program execution proceeds.

A two-byte bit map is used, both in the file format and in memory, to store the attributes of a *type*, a method and a field (state information). Table 4.2 shows a summary of the interpretations of all the bits recognized by a standard JVM. Each bit, when set, is interpreted

bit	Class	Method	Field
1 (LSB)	public	public	public
2	–	private	private
3	–	protected	protected
4	–	static	static
5	final	final	final
6	super	synchronized	–
7	–	–	volatile
8	–	–	transient
9	implementation[1]	native	–
10	interface	–	–
11	abstract	abstract	–
12	–	strictfp	–
13	–	–	–
14	–	–	–
15	–	–	–
16 (MSB)	–	–	–

Table 4.2: Access flags interpretation by the Java VM

as denoting a certain access permission to the *type*, method or field to which this bit map belongs. It may also denote a certain property of the unit.

The use of the access flags is best illustrated by a few examples. The value **0x0621** (bits 1, 6, 10 and 11 are set) when used with a *type* denotes a `public interface` which is automatically set as `abstract`. When used with a method, the value **0x0824** (bits 3, 6 and 12 are set) denotes a `synchronized` and `protected` method in the `FP-strict` floating point mode.

While an *implementation* is compiled to an *interface* with code, it is necessary to differentiate between an *implementation* and an *interface* for two reasons:

1. An *implementation* can contain code, but an *interface* cannot.
2. An *implementation* can inherit from both *interface* and *implementation*, but an *interface* can inherit from an *interface* only.

Bit 9 is chosen to denote an *implementation*. It can be seen from Table 4.2 that the same bit is used to denote a similar attribute in all three levels. For example, bit 1 is used to denote a `public` *type*, a `public` method and also a `public` field, in the respective application. When no common flags can be found, it is intended that the same bit be used at

[1] Inserted to support *implementation* in MCI-Java, but not required by the JVM Specification for standard JVM.

different levels by attributes that will never be applicable in the other levels. For example, bit 6 is used to denote a `super` class and a `synchronized` method because the attribute `super` is not likely applicable to a method, while the attribute `synchronized` is highly unlikely to be applicable to a *type*.

The four most significant bits are not used at all three levels. In order not to interfere with the future development of Java, the four most significant bits will not be used in MCI-Java. In addition, it is very unlikely a *class* would use the modifier `native` and thus, bit 9 is chosen to indicate that the *type* concerned is an *implementation*. The schematic diagram of the entire structure of the modified `.class` file can be found in Appendix B.

4.2 The grammar of MCI-Java

The grammar of MCI-Java can be found in Appendix C. It should be noted that a special sequence of characters reserved for a keyword is shown within a production line as `keyword`. While choices are put inside ( ) and separated by |, optional components are kept within *[]*, as per The Java Language Specification[18].

It is interesting to note that only minimal localized changes to the productions are required to support the new concepts and features; no extensive modifications are necessary.

Type declaration

ClassOrInterfaceDeclaration:
ModifiersOpt (*ClassDeclaration* | *ImplementationDeclaration* | *InterfaceDeclaration*)

ClassDeclaration:
`class` *Identifier* [`extends` *Type*] [`implements` *TypeList*] [`utilizes` *TypeList*]
ClassBody

ImplementationDeclaration:
`implementation` *Identifier* [`extends` *TypeList*] [`implements` *TypeList*]
ImplementationBody

InterfaceDeclaration:
`interface` *Identifier* [`extends` *TypeList*] *InterfaceBody*

Figure 4.1: Modified Java rules for a *type* declaration

The revised rules for a *type* declaration are shown in Figure 4.1. Two new keywords

`implementation` and `utilizes` are introduced to support the new concept of *implementation*. While the use of the keywords `interface`, `implementation` and `class` to represent the three *types* is trivial, the use of the keywords `extends`, `utilizes` and `implements` is explained below:

1. The keyword `extends` is used to express an inheritance relationship between two similar *types*, such as that when one *implementation* inherits from another *implementation*. Of course, this keyword is already used in classic Java for *class* inheritance.

2. The keyword `implements` is used to express the inheritance from an *interface* by a *class* or an *implementation*.

3. The keyword `utilizes` is used to express an inheritance relationship between a *class* and an *implementation*. This keyword is new in MCI-Java.

Multisuper method invocation

Selector:
- *. Identifier [Arguments]*
- *.this*
- *.super SuperSuffix*
- *.new InnerCreator*
- *[Expression]*

SuperSuffix:
- *Arguments*
- *[(Identifier)] . Identifier [Arguments]*

Figure 4.2: Modified Java rules for method invocation

The statement for the newly introduced multisuper method invocation takes the form of `super(preferredParent).methodName()`. In order to accommodate this, an optional component is added in the production for a *SuperSuffix* expression, which is originally used for the super call. The change is reflected in Figure 4.2.

It is important to note that the optional component, the first *Identifier* in the second production for *SuperSuffix*, must be the name of an immediate *superimplementation* of the containing *class* of the method in which this statement is declared. For example, suppose the multisuper call is declared in a *class* which `extends S implements A utilizes B, C`. The multisupercall must be of the form `super(B).m()` or `super(C).m()` where `m()` is the method to be invoked.

4.3 Inheritance and conflict

Code inheritance is handled in Jikes by building an `ExpandedMethodTable`[2] (EMT) for every top-level *type*[3] the compiler encounters. A minor modification to the EMT building algorithm in Jikes is needed to support multiple code inheritance in MCI-Java. The original EMT building algorithm is discussed briefly first so that it is easier to understand the changes made.

4.3.1 The data structure - `ExpandedMethodTable` (EMT)

In Jikes, the EMT is the "in-memory" symbol table for methods. Each *class* has an EMT of its own. Figure 4.3 shows the schematic layout of an EMT. Only those major components that are relevant to this discussion are included in the diagram. A `symbol` in Jikes is a data structure for storing lexical information.

An EMT is an array of linked-lists of `MethodShadowSymbols` each of which is a data structure containing information about a method. The index into the array is the hashed value of a method name (in the form of a character string).

Take, for example, the values included in Figure 4.3 to describe the structure of an EMT. The method names `add`, `foo1` and `foo2` are all hashed to the same value *i*, which is used as an index into the array to locate the linked-list. Each `symbol` in this linked-list is itself the head of another linked-list of `MethodShadowSymbols`, all of them representing methods of the same name but different signatures (the overloaded methods), such as `add(int)`, `add()`, `add(int, int)` and `add(Object)`. Every `MethodShadowSymbol` may have a collection of `MethodSymbols` for methods having the same name and signature; they are the methods in conflict. In Figure 4.3, the conflict pool for the method `add()` has been shown.

For a *class* in the context of classic Java, Jikes starts building an EMT for the current *class* by first entering all its locally declared methods into the table. Constructors, static and private methods will not be entered into the EMT as they cannot be inherited.

After all the locally declared methods are entered into the EMT, elements in the EMT of the immediate *superclass* are added to the local EMT and then followed by those elements in the EMT, of each of the *interfaces* from which this *class* `implements`.

A simple example will help to explain how methods are added to an EMT. Assume that

[2] Denotes an "in-memory" data structure as per Section 1.4.

[3] Any *type* whose declaration is contained in a .java file named using the type name, is a top level *type*. For example, the file Integer.java contains the top-level *class* `Integer`.

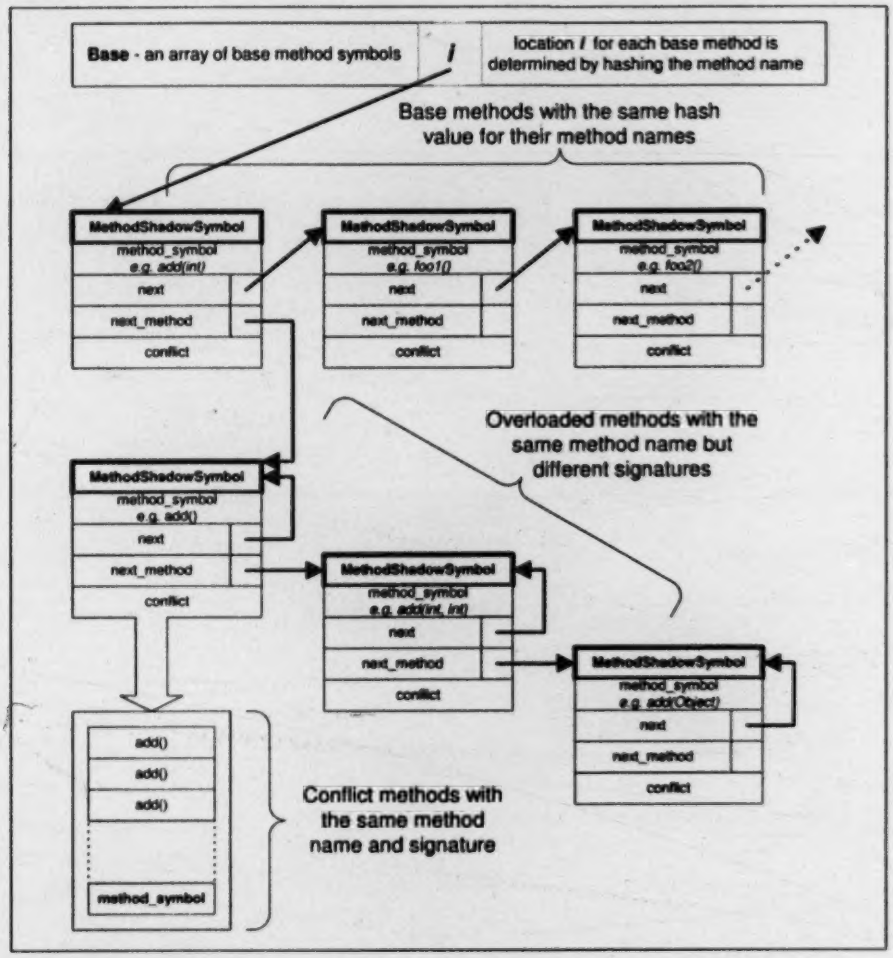


Figure 4.3: Schematic layout of an `ExpandedMethodTable` (EMT)

the `MethodShadowSymbols` for the methods `foo2` and `add(Object)` are not yet in the EMT, as shown in Figure 4.3. To insert a method into the EMT, the method name is first hashed. A search is then conducted in the top-level list of `MethodShadowSymbols` found in the location directed by the hashed value. The method `foo2` is hashed to the value i. A search in the linked-list finds no `MethodShadowSymbol` with the same name, so a new `MethodShadowSymbol` is created for this method and appended to the top-level list.

If a `MethodShadowSymbol` containing a method of the same name as the method to be added exists in the top-level list, a search will be conducted in the list of overloaded

methods. Suppose that the method to be inserted is `add()`. It will be inserted into the pool of methods in conflict because a `MethodShadowSymbol` can be found in the list of overloaded methods that has the same signature. If the method to be inserted is `add(Object)`, it will be appended to the list of overloaded methods because no `MethodShadowSymbol` with the name signature can be found in the list of overloaded methods.

When the merge process is complete, all abstract and overridden methods in the conflict pool will be removed. If any one conflict pool is not empty after this house cleaning stage, an error of ambiguity is reported. If no ambiguity is reported, all the methods in the EMT are methods that are visible in this *class* and will be inherited by all the *subclasses* of the current *class*. If at least one method remaining in the EMT is abstract, the current *class* will be declared abstract.

4.3.2 The revised EMT building algorithm

The major modification to the EMT building procedure to support MCI-Java is the introduction of a conflict resolution procedure using precedence. However, no precedence is given based on lexical order of the *superimplemenatations*. The semantics of MCI-Java are symmetric with respect to the lexical order of the *supertypes*. The three rules are listed below in the order of application:

1. A locally declared method, whether abstract or not, shall have higher precedence than methods declared elsewhere in *types* higher up in the inheritance hierarchy.
2. A concrete method has higher precedence than an abstract method.
3. A method declared in a *type* lower in the inheritance hierarchy will have higher precedence, according to the *relaxed multiple inheritance* policy.

The revised algorithm for building an EMT in MCI-Java is summarized below:

1. All the methods declared in the current *class* are entered into the per-class EMT first.
2. Entries in the EMT of the immediate *superclass* are then added.
3. The EMT of each *implementation* and *interface* inherited by the current *class* will be processed according to the lexical order of the associated *type* in the declaration statement.
4. If no method of the same name and signature as the incoming method exists in the EMT, a new entry will be created for the method to be inserted.

5. When the existing `symbol` contains a locally declared method, all the methods to be added with the same method name and signature will be moved to the conflict pool.

6. If only one of the two methods (the existing one and the incoming one) is abstract, the concrete method will be used as the base method for the shadow symbol and the abstract method will be moved to the conflict pool. Otherwise, the incoming method will be placed in the conflict pool.

7. According to the *relaxed multiple inheritance* policy, the method declared in a *type* which is lower in the inheritance hierarchy tree will have precedence over the other method, if neither one is declared locally.

8. When both the existing base method and the method to be added are declared in the same containing *type*, the incoming method will be ignored so that no error will be signaled when a method is inherited via different inheritance paths.

4.3.3 Revised algorithm verification

Seventeen multiple code inheritance scenarios, which are commonly found in applications programming, are described in Appendix D. Each of these scenarios is now tested against the revised algorithm for building EMT to verify that the modified Jikes is able to identify the ***most specific method*** (§5.3.3) in each situation. While the descriptions of the scenarios are included in the appendix, the pictorial depictions are repeated here, for convenience, in Figure 4.4.

Recall the original algorithm for building the EMT of `C`. It requires that all methods declared in `C` be added first and then followed by the methods contained in the EMT of the direct *superclass* of `C`. The EMTs of all the other direct *supertypes* will be merged to the table in `C` according to the lexical order of the *supertypes* in the declaration line for `C`. However, the result should be independent of the lexical order of these *supertypes* as required by the semantics of MCI-Java, which is symmetric with respect to the order. The results are tabulated in Table 4.3.

It can be seen from Table 4.3 that the compiler behaves as described in Appendix D, according to the semantics of MCI-Java (§3.4). When there is an abstract method in the method shadow symbol and the incoming method is not abstract, as depicted in Figures 4.4(g) and 4.4(h), a swap according to algorithm Rule 6 will move the abstract method to the conflict pool. However, there will be no swapping in Scenarios 11a, 11b and 12a (Figures 4.4(k), 4.4(l) and 4.4(m)), since no abstract methods are involved. Rule 7 will help

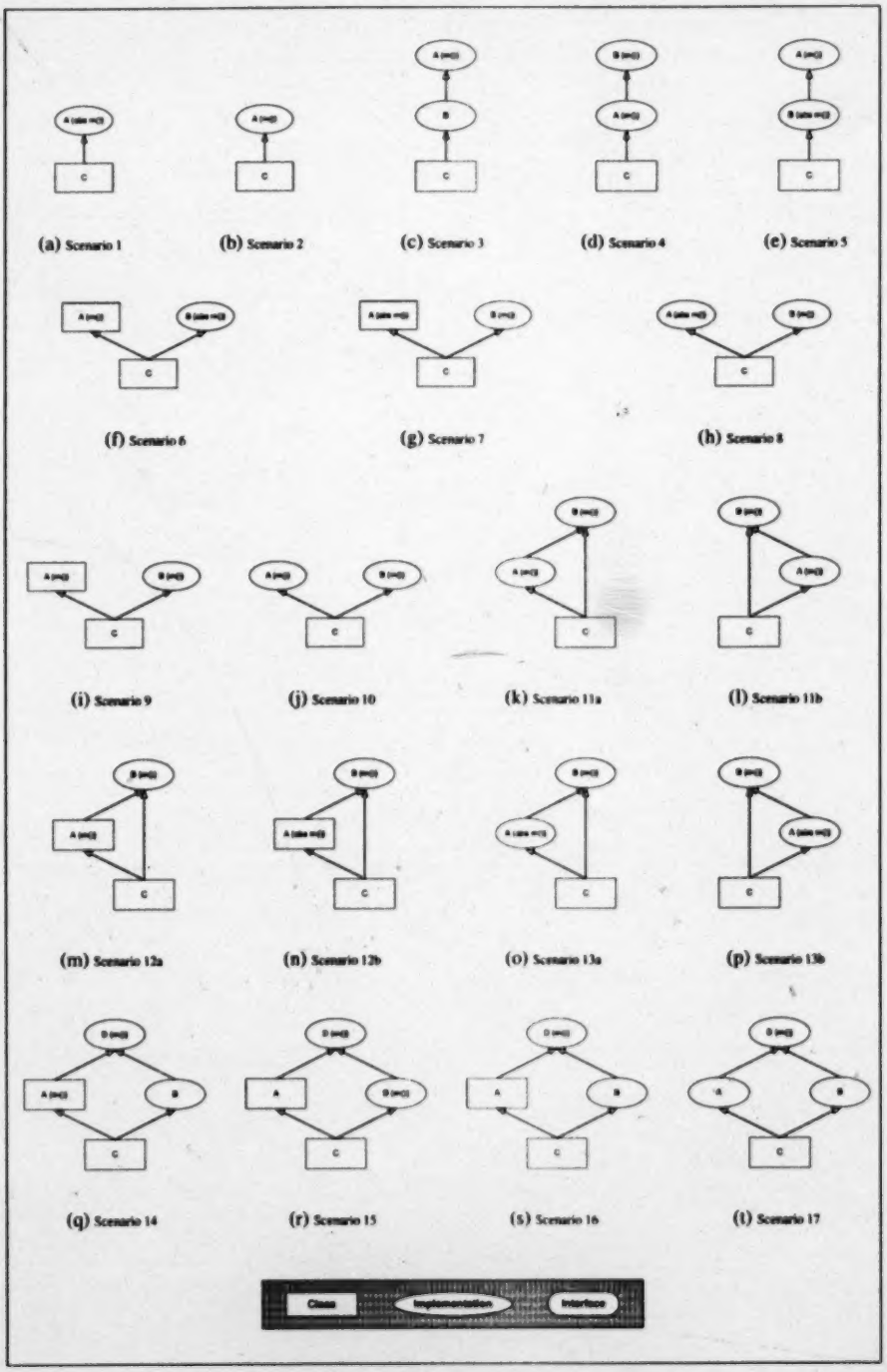


Figure 4.4: Multiple code inheritance scenarios 1 - 17

Scenario	MethodShadowSymbol	conflict table	compiler action
1	abs A::m()	–	declares C abstract
2	A::m()	–	emits A::m() for c.m()
3	A::m()	–	emits A::m() for c.m()
4	A::m()	–	emits A::m() for c.m()
5	abs B::m()	–	declares C abstract
6	A::m()	abs B::m()	emits A::m() for c.m()
7	B::m()	abs A::m()	emits B::m() for c.m()
8	B::m()	abs A::m()	emits B::m() for c.m()
9	A::m()	B::m()	flags ambiguity
10	A::m()	B::m()	flags ambiguity
11a	A::m()	B::m()	emits A::m() for c.m()
11b	A::m()	B::m()	emits A::m() for c.m()
12a	A::m()	B::m()	emits A::m() for c.m()
12b	B::m()	abs A::m()	emits B::m() for c.m()
13a	B::m()	abs A::m()	emits B::m() for c.m()
13b	B::m()	abs A::m()	emits B::m() for c.m()
14	A::m()	D::m()	emits A::m() for c.m()
15	B::m()	D::m()	emits B::m() for c.m()
16	D::m()	–	emits D::m() for c.m()
17	D::m()	–	emits D::m() for c.m()

Table 4.3: Compilation results for scenarios 1 - 17

to give the right resolution for these scenarios. No error of ambiguity will be flagged for Scenarios 16 and 17 (Figures 4.4(s) and 4.4(t)) because of the presence of Rule 8. This is in accordance with the proposed semantics.

4.4 Bytecode generation

No new bytecodes are needed. The existing instruction set of classic Java is sufficient to support MCI-Java.

A new concept (an *implementation*), a new kind of method invocation (the multisuper call) and two new keywords (`implementation` and `utilizes`) are introduced at the source code level. As discussed earlier, an *implementation* is compiled into an *interface* with code and the keyword `utilizes` is used to indicate an inheritance from an *implementation* by a *class*. Thus, no new bytecodes are required for the two new keywords.

One of the criteria imposed on the design of the language extensions proposed in this thesis (§3.1) is the use of the highly optimized component of the JVM for method invocation. For reasons discussed in Section 5.4.4, the existing bytecode *invokespecial* is emitted

for a multisuper call. A minor local change to the method lookup routine is required to extend the semantics of this bytecode. The change made will be discussed in Section 5.4.4

For the `this` expression used in a method definition declared in an *implementation*, the compiler will make a reference to the receiver object. Of course, the *implementation* in which this method is declared will not be instantiated. The reference to the receiver object for this method is reserved for the object instance to which this message is sent at run-time. If the `this` expression is used as an argument to a method, the bytecode *aload_0* will be inserted when the compiler is generating code to push arguments on the stack before a method invocation. This bytecode will, at run-time, cause the reference to the receiver object to be pushed on the stack.

4.5 Summary

MCI-Java introduces a new construct, *implementation*, at the source code level. A new keyword, `utilizes`, is also introduced to express the inheritance relationship between an *implementation* and a *class*. Jikes [4] is modified to support the new features with the extended language semantics.

New production rules are inserted into the original Java grammar in order to enable the compiler to recognize the new features, and some production rules are modified to accommodate the new semantics. A new modifier is added to the binary representation of a *type* to indicate an *implementation*. The procedure to build an EMT, which is used to find the most specific method within an inheritance hierarchy for a method invocation, is modified. The enhanced algorithm is able to resolve conflict in various code inheritance scenarios commonly found in applications programming. The result of the process is as described by the semantics of MCI-Java. The semantics of the bytecode *invokespecial* is extended to cover the newly introduced multisuper method invocation.

While the changes in Jikes are described in this chapter, the modifications required in the Java virtual machine to support MCI-Java semantics will be discussed in Chapter 5.

Chapter 5

Virtual Machine Support

MCI-Java, an extension to classic Java, was proposed in Chapter 3. Changes are made at both the compiler (source code) level (see in Chapter 4) and the VM level. The changes are summarized in Table 5.1 for easy reference.

	Compiler	Virtual Machine
`.class` file	Introduce a new access control flag for a *class* to indicate an *interface* containing code (§4.1)	Modify code verification procedure during *class* loading to recognize an *interface* with code (§5.2)
type declaration	1. Introduce the new keyword `implementation` to indicate a declaration for *code-type* and the new keyword `utilizes` to indicate an inheritance from a *code-type* by a *class* (§4.2) 2. Modify the Java grammar to include the new keywords and introduce an option to accommodate for multisuper method invocation (§4.2)	No changes are required since an *implementation* is an *interface* with code at the VM level
inheritance and resolution	Introduce a modified algorithm for resolving method inheritance and conflict to identify the most specific method for invocation (§4.3)	Extend the use of the virtual method table (VMT) for both a *class* and an *implementation* (§5.3.3) and modify the method lookup routine for the bytecode *invokespecial* (§5.4.4)
method dispatch	Extend the semantics of *invokespecial* to include multisuper method invocation (§4.4)	No changes are made since all method invocations use the existing routines for dispatch

Table 5.1: Summary of changes (VM portion emphasized)

It is necessary to make modifications to both IBM Jikes Compiler and Sun's Java virtual machine to support MCI-Java. Modifications to Jikes have been described in the previous chapter. Here, modifications made to Sun's JVM are discussed.

5.1 The Java Virtual Machine (JVM)

In describing the design principle of a JVM, *The Java Virtual Machine Specification* [23] (JVM Specification) states that,

> The Java virtual machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run-time.

Each Java application runs inside a run-time instance of some concrete implementation of the abstract JVM Specification. The work in this chapter focuses on the implementation by Sun Microsystems Inc., for Java 2 SDK 1.2.2 [3].

5.1.1 The run-time environment

Figure 5.1 shows a simplified schematic run-time memory layout for *class*, *interface*, *implementation* and *object* data. Since Sun's JVM uses the same "in memory" data structure for both *class* and *interface*; the term *class* is generalized in this chapter to cover all three *type* declaring constructs in MCI-Java – namely, *interface*, *class* and *implementation*.

The `heap` is a block of memory shared among all JVM threads. At run-time, every *object* is allocated from the `data area` (region A in Figure 5.1) which is a part of the `heap`. *Object* storage in the `data area` is reclaimed by an automatic storage management system (known as a garbage collector). Each instance is referenced using an `ObjHandle` which is an "in memory" data structure (region B in Figure 5.1) kept in the `data area`. One component of the `ObjHandle`[1] points to the `ObjData`, an "in memory" data structure located in the other part of the `data area` (region C in Figure 5.1), which is holding state values of the object instance. The other component of the `ObjHandle` points to a `ClassClass` structure (an "in memory" per-class structure used in Sun's JVM for holding information about a *class* used in its broad meaning), stored in the `method area` (region D in Figure 5.1) which will not be garbage-collected. If the required `ClassClass` structure is not yet loaded into the run-time environment, the class loader subsystem is responsible for locating and importing the binary data of the concerned *class* (i.e., the `.class`

[1] Denotes an "in-memory" data structure as per Section 1.4.

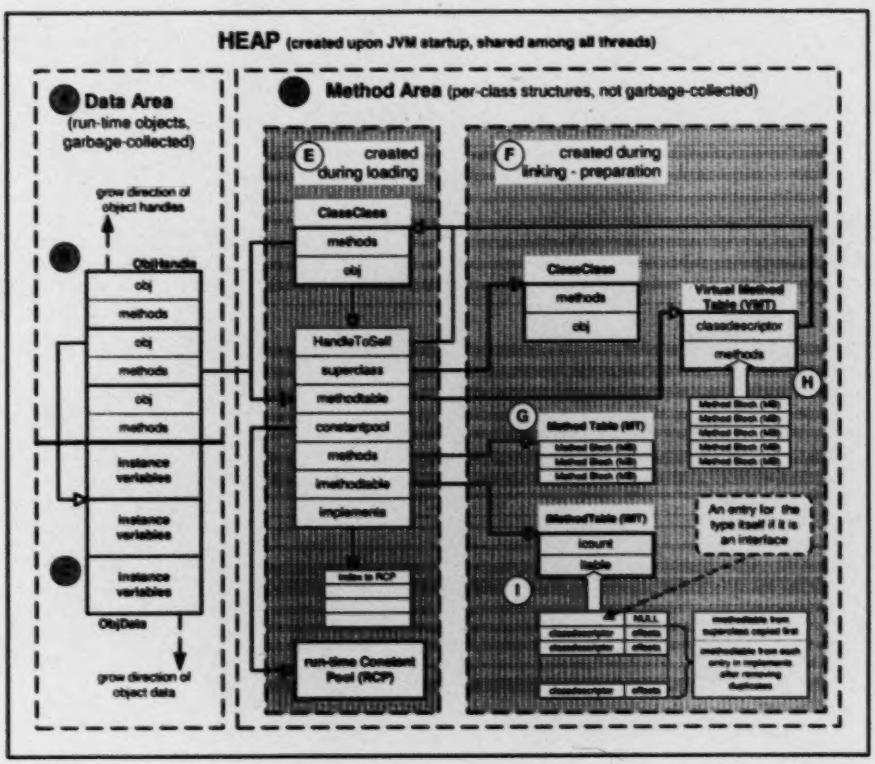


Figure 5.1: Run-time memory layout for *class*, *interface*, *implementation* and *object* data

file). After being verified as well-formed, the *class* data will be linked and become part of the run-time environment.

The symbolic references contained in the `ConstantPool` (a symbol table) of a `.class` file will be transferred to the `run-time ConstantPool` (RCP) (region E in Figure 5.1). Many JVM instructions make symbolic references to the constant pool. Concrete values of the symbols have to be determined before any execution of the instructions can take place. ***Resolution*** refers to the dynamic process of finding these concrete values. When a `ConstantPool` entry is referred to for the first time, it is resolved into an internal reference which will replace the existing RCP entry, corresponding to the constant pool entry being referenced, to ensure that the lengthy resolution process is done only once.

5.1.2 Dynamic per-class structures creation

Upon startup, the JVM does not put all the *classes* into the run-time environment. According to the policy of dynamic loading, a `ClassClass` structure is created for a *class* in the run-time environment only when the *class* is referenced during program execution (referred to as late resolution in the JVM Specification). In summary, the dynamic `ClassClass` structure creation process includes the following stages:

1. Loading

   The JVM will try to find the `.class` file of the *class* in need. Once it is located, the VM will create an internal representation of the *class*, shown in Figure 5.1 marked as region E. Symbolic references are transferred from the `.class` file to the RCP. An exception will be thrown if the required `.class` file cannot be found.

2. Linking

   The loaded *class* will be merged into the run-time environment using the following steps:

   (a) Verification - assuring a well-formed representation (structurally correct) of the *class*,

   (b) Preparation - allocating static storage and internal data structures (region F in Figure 5.1), building the method tables (§5.3) for holding information about methods, and

   (c) Resolution - transforming symbolic references into internal references.

3. Initialization

   The JVM will execute the static initializers and static (class) variable initializers.

5.2 Class data verification

Sun's implementation of the JVM adopts an "on demand" strategy. According to this strategy, a symbolic reference is resolved and a *class* is loaded only when it is needed. After a *class* is loaded, the JVM verifies that its representation is structurally valid. This may cause other *classes* to be loaded. However, the JVM Specification does not require these additionally loaded *classes* to be verified and prepared at that time.

When applied to MCI-Java programs, the original verification procedure used in classic Java will raise a flag for an error in each of the following situations:

1. In the original JVM, all methods declared in an *interface* are implicitly `public` and `abstract`. An "Illegal method modifiers" exception is thrown if a method declared in an *interface* contains code. As discussed earlier, an *implementation* in MCI-Java is compiled to an *interface* containing code. An exception will be thrown when the original verification procedure is used to verify an *implementation*.

2. In the original JVM, the bytecode *invokespecial* is used for a super call. One of the arguments to the bytecode instruction is the name of a *class* (the target class), where a definition of the method to be invoked can be found. The original verification procedure ensures that the target class is not an *interface* and can be found in the superclass chain of the current class. If the condition does not hold, an "Illegal use of nonvirtual function call" exception is thrown. In MCI-Java, the semantics of *invokespecial* has been extended to also denote a multisuper call (§3.3.2) which is an invocation of a method defined in an *implementation* from which the current *class* inherits. In this case, the target class is not a *class* but an *interface* and thus an exception is thrown by the original Java verification procedure.

To support MCI-Java, when an *implementation* is verified after loading, for reasons mentioned above, two minor modifications are applied to the verification procedure:

1. In MCI-Java's VM, all methods declared in an *interface* will only be checked for the modifier `public` but not for the `abstract` modifier. This change ensures that all methods declared in an *interface* and an *implementation* are not private while allowing code in an *implementation* (at the source code level) and an *interface* (at the VM level).

2. In MCI-Java's VM, a search for the target class along the chain of superclasses will be carried out for the bytecode *invokespecial* only when the target class is not an *interface*. This change allows the bytecode to be used for the new multisuper call. However, when it is used for a classic supercall, the target class still has to be a *class* in the superclass chain, so no error-finding is lost.

5.3 Method tables

Sun's JVM uses three kinds of tables – a `Method Table` (MT) (§5.3.2), a `Virtual Method Table` (VMT) (§5.3.3) and an `Interface Method Table` (IMT) (§5.3.4) – to keep information about all the methods of a *class*. The schematic layouts of these

tables can be found in region F in Figure 5.1. The tables are built in the preparation stage of the class linking process. The tables are discussed in the following sections after a brief summary of the `Method Block` (§5.3.1), the basic building unit of the method tables.

5.3.1 `Method Block`

A `Method Block` is the internal representation of a method using the same "in memory" data structure, whether the method is abstract or not. Among all the data stored in a `Method Block`, the most relevant piece is the `FieldBlock` (for which no schematic layout is shown in Figure 5.1) which has the following components:

1. a pointer to the `ClassClass` structure of the containing *class* in which the method is declared – the containing class,
2. a character string representing the method signature – the signature,
3. a character string representing the name of the method – the method name,
4. an access flag masking the modifiers of the method – the modifiers, and
5. an integer value representing the ordinal of the method within an array of `Method Blocks` – the offset.

In addition to the `FieldBlock`, there is a code pointer to the memory location where the bytecode sequence of that method is stored. The code pointer is null when the method is an `abstract` one, because the method has no code.

5.3.2 `Method Table` (MT)

A `Method Table` exists in every `ClassClass` structure whether the concerned *type* is a *class* or an *interface*. It is the part of the `ClassClass` structure that stores information about every method declared in this *class* (in its broader meaning), including the constructors, private and static methods. A simplified schematic layout of an MT and its link with the `ClassClass` structure is shown in region G of Figure 5.1. Each `Method Block` (§5.3.1) will point to the current `ClassClass` structure itself as the containing class.

If the current *type* is an *interface*, all methods in the MT are `public` and `abstract`. Each `Method Block` has an offset value reflecting the order of appearance of the method in the *interface* declaration. If the current *type* is a *class*, each `Method Block` has an

initial offset value of 0. If the method is a public instance method, this offset value will be changed to the slot number of the *slot* (a VMT entry) in which this method is stored in the VMT (§5.3.3) when the VMT is built.

Since no explicit rule exists to restrict a `Method Block` found in the MT of an *interface* from containing code, the same data structure can also be used to store information about methods declared in an *implementation*, whether the methods are abstract or not.

5.3.3 `Virtual Method Table` (VMT)

The original setup

The VMT in Java is similar in nature to the `Virtual Function Table` (VFT) in C++. It helps to identify the method to be dispatched at run-time. Each slot holds a reference to an instance method definition visible in the current *class*. Since an *interface* is used in Java for specifying behavior and contains no concrete method implementation, no VMT is built for an *interface*. A VMT is found only in the `ClassClass` structure of a *class*. Region H in Figure 5.1 shows the structure of a VMT and the way in which it is linked to the `ClassClass` structure.

Figure 5.2 shows the VMT details of two *classes* taken from the `java.io` package. `DataInputStream` extends `FilterInputStream` which inherits `InputStream`, a subclass of `Object`. Entries in the VMTs help to explain the VMT building procedure. These *classes* belong to the inheritance hierarchy depicted in Figure 6.6.

In OOP, *classes* related in an inheritance relationship are permitted to provide code for the same method. The method definition provided in the *subclass* is called the ***most specific method***. The most specific method overrides all the implementations for the same method, previously defined in all the *superclasses* of the *class* in which the most specific method is declared. The most specific method will remain in effect until it is overridden by another method definition in a *subclass* of the current *class*.

The process for building the VMT for a *class* allows the most specific method to override all previous definitions for the same method. It starts by copying all the slots from the VMT of the immediate *superclass*. Once VMT copying is completed, the public instance methods declared in the current *class* will be entered into the VMT. It can be seen from Figure 5.2 that 11 methods (offset values between 1 and 11, inclusive) are copied from `Object` to `FilterInputStream` via `InputStream`. In addition, the 9 methods declared in `FilterInputStream` (offset values between 12 and 20, inclusive) are copied to the VMT of `DataInputStream`.

```
**** java/io/InputStream ****
Superclass: java/lang/Object


**** java/io/FilterInputStream ****
Superclass: java/io/InputStream
Details of VMT with 20 entries:

offset   Class  Method Signature
------ -------  ------------------
     1  Object  clone () LObject;
     2  Object  equals (LObject;) Z
     3  Object  finalize () V
     4  Object  getClass () LClass;
     5  Object  hashCode () I
     6  Object  notify () V
     7  Object  notifyAll () V
     8  Object  toString () LString;
     9  Object  wait () V
    10  Object  wait (J) V
    11  Object  wait (JI) V
    12     FIS  available () I
    13     FIS  close () V
    14     FIS  mark (I) V
    15     FIS  markSupported () Z
    16     FIS  read () I
    17     FIS  read ([B) I
    18     FIS  read ([BII) I
    19     FIS  reset ()V
    20     FIS  skip (J)J



Legends:

  FIS = java/io/FilterInputStream
  DIS = java/io/DataInputStream
Class = Declaring class


Descriptors used:
B - signed byte
C - character
D - double
F - float
I - integer
```

```
**** java/io/DataInputStream ****
Superclass: java/io/FilterInputStream
Details of VMT with 35 entries:

offset   Class  Method Signature
------ -------  ------------------
     1  Object  clone () LObject;
     2  Object  equals (LObject;) Z
     3  Object  finalize () V
     4  Object  getClass () LClass;
     5  Object  hashCode () I
     6  Object  notify () V
     7  Object  notifyAll () V
     8  Object  toString () LString;
     9  Object  wait () V
    10  Object  wait (J) V
    11  Object  wait (JI) V
    12     FIS  available () I
    13     FIS  close () V
    14     FIS  mark (I) V
    15     FIS  markSupported () Z
    16     FIS  read () I
    17     DIS  read ([B) I
    18     DIS  read ([BII) I
    19     FIS  reset () V
    20     FIS  skip (J) J
    21     DIS  readBoolean () Z
    22     DIS  readByte () B
    23     DIS  readChar () C
    24     DIS  readDouble () D
    25     DIS  readFloat () F
    26     DIS  readFully ([B) V
    27     DIS  readFully ([BII) V
    28     DIS  readInt () I
    29     DIS  readLine () LString;
    30     DIS  readLong () J
    31     DIS  readShort () S
    32     DIS  readUTF () LString;
    33     DIS  readUnsignedByte () I
    34     DIS  readUnsignedShort () I
    35     DIS  skipBytes (I) I


J - long integer
S - signed short
V - void
Z - boolean
L<classname>; - a refernece
[ - one array dimension
```

Figure 5.2: Virtual method table (VMT) entries

For each entry in the MT that is a public instance method (herein referred to as the current method), a search is conducted in the newly created VMT to look for a slot which contains a pointer to a method with the same signature as the current method. If none exists in the VMT, a new slot is appended to the VMT for the current method, with a pointer to the `Method Block` in the MT holding information about the current method. The offset value of this `Method Block` is then changed to the new slot number. If an existing slot

in the VMT is found to point to a method with the same signature as the current method, the offset value of the `Method Block` in the MT for the current method is set to the slot number just found. The pointer in that slot of the VMT is made to point to the method block in the MT for the current method. This is literally a method override by the most specific method in the current *class*. In the VMT of `DataInputStream`, methods with offset value 17 and 18 are examples of method overriding, while methods with offset values between 21 and 35, inclusive, are those added in `DataInputStream`.

At the end of the process, all `Method Blocks` for the locally declared public instance methods in the MT will have their offset values set to the corresponding slot number in the VMT. A zero value indicates that the method is not inheritable. These include constructors, static and private methods. At the same time, each slot in the VMT points to the most specific method implementation visible in the current *class*.

While new slots are added in the VMT of the *subclasses*, and some old slots are overridden, the offset value (the slot number) for a method with the same name and signature remains the same in all the *classes* concerned. This consistency in slot number, reflected in Figure 5.2, provides room for optimization. It is employed in Sun's implementation of the JVM in the form of quick bytecode, which will be discussed in Section 5.4.4.

The modified setup

To support *implementation* in MCI-Java, it was decided to build a VMT for an *interface*. This is because an *implementation* in MCI-Java is compiled to an *interface* at the VM level. Similar to the situation with a *class*, it is necessary to maintain a pool of the most specific methods visible in an *implementation*. The most efficient way is to use a mechanism analogous to the one used in *classes*. As the "in memory" data structure used in Sun's JVM does not prohibit it from being used for a construct other than a *class*, building a VMT for an *implementation* becomes a natural solution to the problem.

Whether it is a *class* or an *implementation*, the same procedure is used to build the VMT. Entries in various tables are added to the new VMT in the following order:

1. All entries in the VMT of the *superclass* are copied in the original order to the new VMT first (this step is skipped for an *implementation*).

2. For each of the *superimplementations*, all methods in its VMT are added to the new VMT. Each *superimplementation* is processed according to its lexical order of appearance.

3. Only public, non-abstract instance methods from the local MT are added to the new VMT.

When a method is added to the new VMT, and no existing slot in the VMT contains a method with the same name and signature as the one to be added (referred to as the new method), a new slot is created for the new method. If another method with the same name and signature as the new one already exists in the VMT, a potential inheritance ambiguity occurs. The following rules are applied to resolve the conflict:

1. If the new method is declared locally, it overrides the existing one. This rule is used in classic Java when building VMT for a *class*.

2. If both methods have the same *implementation* as the containing class, they are inherited from the same source via different paths. The new method is ignored.

3. If the two methods have different containing classes and an ancestor/descendant relationship can be established between them, the method declared in the *type* which is lower in the inheritance hierarchy will be kept in the new VMT while the other one is dropped, according to the *relaxed multiple inheritance* policy.

4. If the two methods are declared in different *types* and no ancestor/descendant relationship can be established between them, the MT of the current *type* will be searched for a method with the same name and signature. If such a method exists, the incoming method is ignored because it will be overridden when the locally declared method is added to the new VMT. If no such method exists in the current MT, an ambiguity exception is raised.

When the VMT is built for a *class*, the offset value in the `Method Block` of the locally declared method will be set to the number of the slot in which the method is inserted into the VMT. However, the offset value is not set for the local methods if the VMT is being built for an *implementation*, for reasons to be discussed in the next section.

5.3.4 `Interface Method Table` (IMT)

The slot in which an interface method is inserted into a VMT depends on the number of *interfaces* from which the current *class* inherits and the number of methods declared in each of these *interfaces* as well as that of the current *class*. Therefore, consistency in the slot number cannot be found in methods declared in an *interface*, in contrast to the consistent slot number associated with each instance method declared in a *class* (§5.3.3). Determining

```
**** Interface java/io/DataInput ****
Details of MT with 15 entries:

offset Method
------ ------
     0 readBoolean ()Z
     1 readByte ()B
     2 readChar ()C
     3 readDouble ()D
     4 readFloat ()F
     5 readFully ([B)V
     6 readFully ([BII)V
     7 readInt ()I
     8 readLine ()Ljava/lang/String;
     9 readLong ()J
    10 readShort ()S
    11 readUTF ()Ljava/lang/String;
    12 readUnsignedByte ()I
    13 readUnsignedShort ()I
    14 skipBytes (I)I

**** Class java/io/DataInputStream ****
This class implements 1 interfaces
VMT with 35 entries

Details of IMT with 1 interfaces:

                     Interface  Offsets
-----------------------------  -----------
             java/io/DataInput  21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

**** Class java/io/RandomAccessFile ****
This class implements 2 interfaces
VMT with 49 entries

Details of IMT with 2 interfaces:

                     Interface  Offsets
-----------------------------  -----------
            java/io/DataOutput  36 37 38 39 40 41 42 43 44 45 46 47 48 49
             java/io/DataInput  19 20 21 22 23 24 25 26 27 28 29 30 31 32 35


Descriptors used:                       J - long integer
B - signed byte                         S - signed short
C - character                           V - void
D - double                              Z - boolean
F - float                               L<classname>; - a refernece
I - integer                             [ - one array dimension
```

Figure 5.3: Interface method table (IMT) entries

the slot number after a method is located in the MT of an *interface* is non-trivial. An IMT helps to locate the slot number for an interface method at run-time.

A simplified schematic layout of an IMT can be found in region I of Figure 5.1. An IMT contains an integer value, `icount`, representing the total number of *interfaces* listed in the `itable`. The `itable` is an array, each entry of which represents exactly one unique *interface* extended (by an *interface*) or implemented (by a *class*) by the current *type*. In

every entry of the `itable`, there is an array of offset values and a `classdescriptor` pointing to the `ClassClass` structure of the *interface* concerned. If the current *type* is a *class*, the first entry of the `itable` is left blank. If the current *type* is an *interface*, the first entry of the `itable` points to the `ClassClass` structure of the *type* itself. Every element of the offset array corresponds to a method in the MT of the *interface*. Each offset value indicates the slot number in the current VMT where an implementation of the corresponding interface method can be found. However, the offset values will just be the ordinal of the corresponding method within the MT of the *interface*, if the IMT belongs to an *interface* which has no VMT.

Figure 5.3 helps to explain the structure of an IMT. In this figure, the 15 methods declared in `DataInput` (an *interface*), found in the `java.io` package, are listed according to the order of appearance of each method in the MT. `DataInputStream` and `RandomAccessFile`, also from the same package, are the two *classes* implementing `DataInput` (see Figure 6.6 for a depiction of the inheritance hierarchy). It can be seen that the corresponding slot numbers in the respective VMTs are different for the same method in `DataInput`. For example, the first method in `DataInput` is `readBoolean()`. Its slot number in the VMT of `DataInputStream` is 21, while that in the VMT of `RandomAccessFile` is 19.

The process of building an IMT for a *class* does not change with the introduction of *implementations* and the accompanying VMTs in MCI-Java. New methods inherited from the *superimplementations* are only appended to the VMT of the current *class* at the end, and are not inserted in the middle of the table.

To build the IMT of a *class*, the IMT of the *superclass* is copied in its entirety to the newly built one. This step is skipped when building the IMT of an *interface*. Then, all the *superinterfaces* of the current *class* will be processed in turn. If an entry already exists in the IMT for an *interface* with the same name as the one to be processed, the *interface* will not be processed. For each method contained in the MT of the *interface*, a search will be conducted in the current VMT. When a slot containing a method with the same name and signature is found, the slot number will be written to the offset array in the corresponding position. For example, the first entry in the MT of `DataInput` is the method `readBoolean()`. In the VMT of `DataInputStream` (Figure 5.2), the method `readBoolean()` is found in slot 21. Therefore, 21 is entered as the first entry in the offset array for `DataInput` in the `itable` of the IMT of `DataInputStream`. When the IMT is built for `RandomAccessFile`, the slot number for the same method

`readBoolean()` becomes 19 and so the first entry in the offset array for `DataInput` in the `itable` of the IMT of `RandomAccessFile` is 19 instead of 21. Since the slot numbers for methods in the MT of an *interface* are not in a consistent pattern as that found in the instance methods declared in a *class*, the offset value of the `Method Block` in the MT of an *interface* (`DataInput` in this case) is not changed so that the offset value in the `Method Block` always gives the index into the array of offsets in any `itable`. This explains why the offset value of a `Method Block` in the MT of an *implementation* is not set to the slot number after the method is inserted into a VMT (§5.3.3).

While the IMT building process is not changed, there will be a minor adjustment to the procedure to support *implementation* in MCI-Java. Details are discussed in the next section.

5.3.5 Setting offset values in `Method Blocks`

As revealed in Section 5.3.3, instance methods declared in a *class* with the same name and signature will have the same slot number. This consistency allows the offset value (slot number) to be stored in the `Method Block` representing the method. However, the consistency in slot number does not exist in methods declared in *interfaces*. An ordinal value has to be stored in the `Method Block` to help to locate the slot number using the IMT.

It was determined in Section 3.1 that the highly optimized bytecode execution routines should be reused in MCI-Java. The routines assume and require that the necessary information for bytecode execution (including the slot number which guides the retrieval of an execution method block from a VMT) is contained in the `Method Block`. This special feature becomes a problem for the methods declared in an *implementation* because each `Method Block` stores the ordinal number instead of the slot number in the offset field.

The problem can be solved using a clone of the `Method Block` concerned. In classic Java, each element of the VMT is actually a pointer to the `Method Block` containing the most specific method which is stored in the MT of the containing class. In MCI-Java, the VMT element still points to a `Method Block` in the MT of the method's containing class, if it is a *class*. This is the same as in classic Java. However, when the containing class is an *implementation*, a clone of the `Method Block` for the most specific method is created in the `method area` (region F in Figure 5.1) to hold the slot number, which will remain the same for all the *classes* inheriting from the current *class* once the `Method Block` is entered into the current VMT. The pointer in the VMT will point to the clone instead of to the original `Method Block` in the MT of the method's containing class. The

clone helps to provide all the necessary information required by the bytecode interpretor. Different clones are used in different inheritance trees in which the slot numbers assigned are usually different from each other. Such an arrangement makes it possible for different offset values (slot numbers) to coexist in the same run-time environment for the same method implementation.

5.4 Method lookup and invocation

5.4.1 Overview

The JVM Specification has classified all run-time method invocations into four main categories with different bytecodes used in the instruction in each case, as described below:

1. *invokestatic* - to invoke a static (class) method which requires no receiver object for invocation.

2. *invokespecial* - to denote a super call or to invoke a private method of the current *class*.

3. *invokeinterface* - to invoke a method whose receiver object has a static type that is an *interface*.

4. *invokevirtual* - to invoke a method in scenarios not covered by the other three categories (the most widely used invocation in Java programs).

Each invocation instruction has one of the four invocation bytecodes followed by at least two more bytes of information. These two bytes, when combined together, give the index to the `ConstantPool` where a reference to the method to be invoked can be found. Whenever a method is invoked, a `Frame` is created to store data – partial results as well as the return value. Every method invocation is a four-step procedure, described as follows:

1. The `ConstantPool` entry is dereferenced to obtain the reference to the ***target class*** which is, at compile time, the containing class of the method intended to be invoked. In addition, the reference to the details of the method (method name and signature) is found at the same time. It is important to note that the target class is very often not the *class* to which the receiver object belongs.

2. A search for the resolved `Method Block` (RMB) is conducted starting from the MT of the target class.

3. The original invocation bytecode is quick-ed (§5.3.3) for highly optimized bytecode execution.

4. The actual execution `Method Block` (EMB) is retrieved and executed using the routine for the quick bytecode, and the information retained when the bytecode is quick-ed.

It is important to note that the last step (bytecode execution) is carried out using a highly optimized routine. Changes should not be made to this part of the VM to avoid affecting its performance (§3.1).

5.4.2 Dereferencing the `ConstantPool` entry

In this section, the techniques used in classic JVM to dereference a method invocation instruction are described. In MCI-Java, regardless of whether the method to be invoked is declared in a *class* or in an *implementation*, the same procedure is used; no change is necessary.

Symbolic references in the run-time `ConstantPool` (RCP) are resolved into internal references to the corresponding "in-memory" data structures when they are used the first time. The dereferencing process is best illustrated by using an example. Figure 5.4 shows an excerpt of the `ConstantPool` for `DataInputStream`. The actual `ConstantPool` contains more entries than those shown in the diagram. However, only those related to the discussion that follows (e.g., entries at indices 1, 6, 30, etc.) are shown.

Take for example a method invocation instruction "*invokevirtual 32*". This will cause the method referenced by RCP entry 32 to be invoked. From Figure 5.4, the `ConstantPool` entry at index 32 is a `MethodReference` which is a composite of two indices. The first index, 6, refers to the target class. The unresolved entry at location 6 points to 97 for the name of the target class which is "InputStream". Once the name is known, the `ClassTable` (an "in-memory" list of loaded *classes* maintained by the JVM) is queried to locate the run-time `ClassClass` structure for `InputStream`. The pointer to this `ClassClass` structure then replaces the original entry at location 6 of the RCP. Any subsequent instruction that refers to RCP entry 6 will obtain a pointer to the `ClassClass` structure of `InputStream` without going through the entire dereferencing and lookup process.

The second index at RCP entry 32, which is 53, refers to a `NameAndType` index, giving information about the method to be invoked. The entry at RCP 53 is again a composite

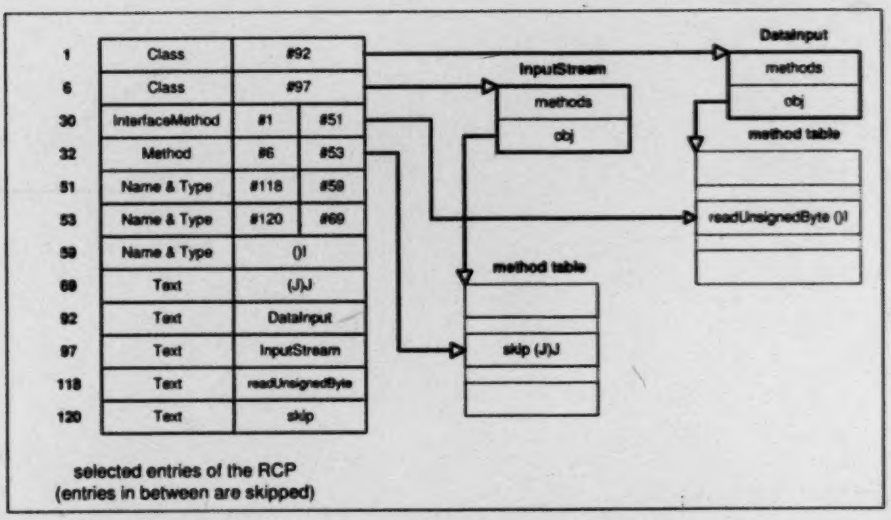


Figure 5.4: Excerpt of the run-time `ConstantPool` (RCP) for `DataInputStream`

item. Its first index (120) points to an RCP entry which stores the method name "skip", while the method signature "(J)J" is kept in the entry referred to by the second index (69). Therefore, the method referred to at RCP entry 32 is, in its fully expanded symbolic form, `InputStream::skip(J)J`. Once these are known, the search process for the RMB starts from the MT of the target class, which is `InputStream` in this case. This search process is described in more detail in the next section. When the RMB is located, the internal pointer to this RMB replaces the original entry at location 32. Similarly, any subsequent references to this RCP entry 32 will obtain the internal reference to the method `skip(J)J`, contained in `InputStream`, without further dereferencing and lookup.

Take for example another instruction, "*invokeinterface 30*". By the same process described above, the method referenced by RCP entry 30 is resolved into a reference to the interface method `DataInput::readUnsignedByte()I` declared in `DataInput`, which is itself an *interface*.

5.4.3 Method lookup

The original method lookup procedure used in classic Java can be applied to MCI-Java without any modifications, except for the multisuper call (§5.4.5). The search for the RMB starts in the MT of the target class. The search will follow one of the two paths described below:

1. When the bytecode is *invokeinterface*, the target class itself is an *interface* and the the search makes use of the IMT of the target class. The `itable` of the IMT contains the list of *superinterfaces* from which the target class `extends`. Recall that the first entry in the `itable` is the target class itself (§5.3.4). By looking up the MT of each *interface* contained in the `itable`, following the order of appearance within the `itable`, the process basically searches the MT of the target class first, followed by that of each and every *superinterface* listed in the `itable`.

2. When the bytecode is not *invokeinterface*, the target class itself is a *class* and the search starts in the MT of the target class. If no matching `Method Block` is found, the search will continue in the MT of the *superclasses* until the root class `Object` is reached. If no RMB can be found at this stage, the search will be conducted in the VMT of the target class. An exception will be raised if nothing is found in the VMT. The only time that a method can be found in the VMT of the target class, but not in the MTs of the target class and its *superclasses*, is when the RMB is a Miranda method [14, 16].

It is important to note that the search will stop when a matching `Method Block` is found. The JVM is only responsible for enforcing access controls (e.g., limited access to private methods) after a `Method Block` is located, but not during the lookup process. In other words, the lookup process will search every method it can reach without paying attention to the accessibility issue. The first `Method Block` encountered with the same name and signature will be selected and the search process will terminate with the first `Method Block` located. If the selected method is a constructor, an error will be raised and the program execution halted. The same thing will occur when the selected method is not accessible by the receiver object (e.g., a private method accessed by an instance of a *class* different from the method's containing class). Otherwise, the selected `Method Block` will be returned as the RMB. A pointer to the RMB replaces the original RCP entry and is put into a `ConstantPool` within the method invocation `Frame` for further process as described in Section 5.4.4.

It was said at the beginning of this section that the same process can be applied to MCI-Java without requiring any change. The discussion that follows explains how the original process can deal with a situation when a method whose containing class is an *implementation* is to be invoked. When the method is invoked using a bytecode other than *invokeinterface*, the procedure will start the search in the MT of the target class which is

the *implementation*. Of course, the RMB can be located very easily. If the bytecode is *invokeinterface*, the search begins in the *interface* which is the first entry in the IMT of the target class, which is the *implementation* itself. Again, the RMB can also be located in the MT of the *implementation*. Thus, no change is needed for the method lookup procedure for use in MCI-Java.

Original bytecode	Context	Quick-ed bytecode
invokestatic	always quick-ed to the same quick bytecode	*invokestatic_quick*
invokevirtual	the slot number is contained in one byte (slot number < 256)	*invokevirtual_quick*
	slot number ≥ 256	*invokevirtual_quick_w*
	the static type of the receiver object is `Object` (e.g., an array or an element from a container	*invokevirtualObject_quick*
invokeinterface	always quick-ed to the same quick bytecode	*invokeinterface_quick*
invokespecial	the RMB contains a private method or the same method definition is visible in the current class and the superclass in the case of a supercall	*invokenonvirtual_quick*
	different method definitions are visible in the current class and the superclass	*invokesuper_quick*

Table 5.2: Conversion from execution bytecode to quick-ed bytecodes

5.4.4 The quicking process and code execution

This section is important in understanding the modifications required to support the new multisuper call introduced in MCI-Java. To redirect code execution to the right optimized routine, the original bytecode will be replaced by one of the seven quick bytecodes listed in Table 5.2, according to the context in which the execution is invoked. These contexts are also included in the same table. After the RMB is identified by the lookup process described in Section 5.4.4, crucial data are kept in a `ConstantPool` within the method invocation `Frame`. The data will be used in the subsequent optimized execution process, as described below.

invokestatic_quick

The bytecode *invokestatic* is used to invoke a static method, which is the same as a procedure call in C and thus has no receiver object. No reference will be provided for the receiver object (`this`) during execution. A pointer to the RMB is stored in the `ConstantPool` within the method invocation `Frame`. The optimized routine will use the stored RMB as the EMB and invoke the method directly.

invokevirtual_quick and *invokevirtualObject_quick*

The offset value is retrieved from the RMB and stored in the `ConstantPool` within the method invocation `Frame`. The optimized routine will use the stored value as the slot number to retrieve the most specific method from the VMT of the current class as the EMB for execution.

invokevirtual_quick_w

A pointer to the RMB is stored in the `ConstantPool` within the method invocation `Frame`. The optimized routine will use the offset value stored in the RMB as the slot number to retrieve the most specific method from the VMT of the current class as the EMB for execution.

invokeinterface_quick

A pointer to the RMB is stored in the `ConstantPool` within the method invocation `Frame`. Recall that a `Method Block` contains a field which points to the method's containing class (§5.3.1) which should be an *interface* for the RMB, for the bytecode *invokeinterface*. All methods in the MT of an *interface* have an offset value, which is the ordinal within the MT.

The optimized routine will invoke a search in the IMT of the current class for the *interface*, which is the same as the method's containing class for the RMB stored in the `ConstantPool` within the method invocation `Frame`. Once the *interface* is located in the IMT, the offset value retrieved from the RMB is used as an index to the array of offsets to obtain the slot number for retrieving the EMB from the current VMT for execution.

invokesuper_quick

The offset value is retrieved from the RMB and stored in the `ConstantPool` within the method invocation `Frame`. The optimized routine will use the stored value as the slot

number to retrieve the most specific method from the VMT of the superclass as the EMB for execution.

invokenonvirtual_quick

A pointer to the RMB is stored in the `ConstantPool` within the method invocation `Frame`. The optimized routine will use the stored RMB as the EMB and invoke the method directly.

Bytecode for multisuper call

Ultimately, only a modified *invokespecial* is suitable for a multisuper call. The semantics of multisuper call require that the method to be invoked must be declared in an *implementation* which is inherited by the current *class* either directly or indirectly. Among the four original execution bytecodes, only *invokestatic* and *invokespecial* are replaced by quick-ed bytecodes that do not retrieve the EMB from the current VMT. Since *invokestatic* does not have the provision for the `this` pointer, which is needed by a multisuper call, *invokespecial* is therefore chosen for multisuper call. However, the method lookup procedure for *invokespecial* only searches the MT along the chain of *superclasses*, not those of the *superinterfaces*. A change to the procedure is required to accommodate the extended semantics of *invokespecial* to include multisuper call. The change is described in the following section.

5.4.5 Multisuper call

Method lookup

In classic Java, the method lookup procedure starts the search for the RMB in the MT of the target class. When the RMB is located, the offset value stored in the `Method Block` will be used to retrieve the method referenced in the corresponding slot in the VMTs of both the current *class* and the *superclass*. If the two methods are the same, the quick bytecode *invokenonvirtual_quick* is used; otherwise, *invokesuper_quick* is used.

In MCI-Java, *invokespecial* is also used for multisuper call. In this case, the target class is an *implementation* at the source code level and an *interface* at the VM level. The offset value stored in the `Method Block` is only an index to the array of offset values in the `itable` (§5.3.3) of the IMT. The offset value is not a true slot number in the VMT. To avoid choosing a wrong EMB, the original procedure is modified as described below.

When the RMB contains a private method or the target class is a *class* and not an *implementation* (i.e., an *interface* with code), the original procedure is followed because the bytecode is used for its original semantics in classic Java for a super call, or to invoke a private method. When the target class is an *implementation*, the bytecode is used for a multisuper call. The procedure described in Section 5.4.4 is redirected to a new lookup routine.

In the original procedure, a `Tag` is used in the RCP to indicate whether or not the method to be invoked is an interface method. The `Tag` is retrieved by the lookup procedure as the value of a temporary variable. If it is the value for an interface method, as in the case of *invokeinterface*, the search for the method will be conducted in the MT of the *interfaces* listed in the IMT of the current *class* only. The new routine makes use of this special feature.

Normally, *invokespecial* comes with a method reference `Tag`, instead of an interface method reference `Tag`. Once the target class is known to be an *interface* (for an *implementation*) after resolution, the value of the temporary variable is changed to the value for an interface method reference, so that the routine for searching only the IMT of the target class is used. The RMB located in one of the *superimplementations* is put in the run-time `ConstantPool` replacing the original symbolic entry, and a pointer to the RMB is put in the `ConstantPool` within the method invocation `Frame`, as well. The quicked bytecode *invokenonvirtual_quick* replaces the original bytecode *invokespecial*. Recall that *invokenonvirtual_quick* uses the stored RMB as the EMB for direct execution. Thus, a method declared in one of the *implementations* in the *superimplementation* tree is executed as required by the extended semantics of *invokespecial*.

5.4.6 Example scenarios

In this section, it is shown that the modified method lookup mechanism is able to locate the correct EMB for execution. Several inheritance scenarios, as shown in Figure 5.5, are used for this empirical verification. These scenarios are taken from the collection discussed in Appendix D, using the same scenario numbers.

Table 5.3 shows the verification results. It is assumed that `c` is an instance of `C` while `a` and `b` are object instances having static types of `A` and `B` respectively. With the help of the VMTs built for *implementations*, and the modified resolution procedure, the EMB located for each scenario and call combination follows what is expected using the new semantics of MCI-Java.

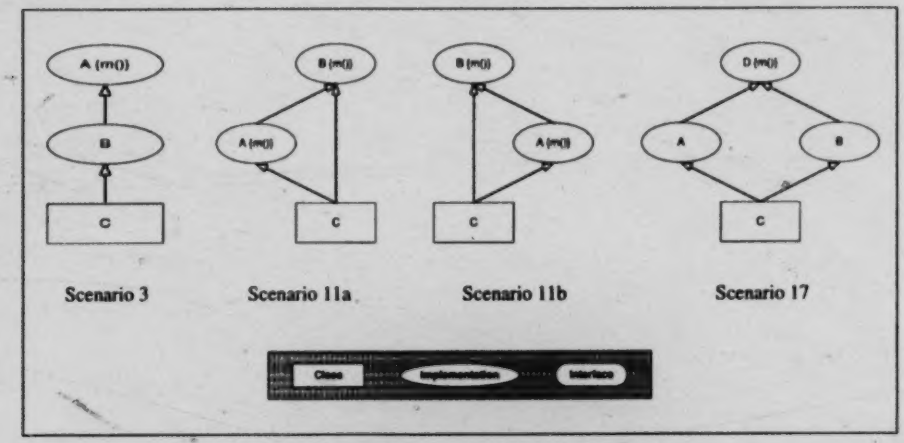


Figure 5.5: Selected multiple code inheritance scenarios from Figure 4.4

5.4.7 Recompilation of a supertype

The Java platform permits the recompilation of an individual module (i.e., a .java file) without requiring the recompilation of all other related declarations. According to the JVM Specification, all JVM implementations should be able to take into account all the changes as a result of the recompilation, and still be able to invoke the most specific method at run-time for each invocation bytecode. The following discussions (referring to scenarios depicted in Figure 5.5) are intended to verify that the dynamic semantics are followed in MCI-Java, since no *subclass* is forced to be recompiled after a *superclass* is recompiled.

Scenario 3

Suppose that `B` is recompiled to insert a new overriding implementation for `m()`. The slot in the VMT of `C` for the method `m()` should point to a clone of the `Method Block B::m()`. The dynamic semantics is still followed for all three instance method invocation bytecodes.

For the instruction *invokevirtual* `A::m()`, the method look up process finds `A::m()` as the RMB whose offset value allows the JVM to locate `B::m()` in the VMT of `C` as the EMB, and thus it is dispatched in response to the message `c.m()`.

The search process for *invokeinterface* `A::m()` starts from the target class `A`. The search locates `A::m()` as the RMB. Offset lookup in the IMT of `C` results in an index to the VMT of `C`. The VMT entry points to `B::m()`, and thus it becomes the EMB and is executed.

Scenario	Compiler Output	Execution Method Block (EMB)
	Message sent: `c.m()`	
Scenario 3	*invokevirtual* `A::m()`	`A::m()`
Scenario 11a	*invokevirtual* `A::m()`	`A::m()`
Scenario 11b	*invokevirtual* `A::m()`	`A::m()`
Scenario 17	*invokevirtual* `D::m()`	`D::m()`
	Message sent: `a.m()`	
Scenario 3	*invokeinterface* `A::m()`	`A::m()`
Scenario 11a	*invokeinterface* `A::m()`	`A::m()`
Scenario 11b	*invokeinterface* `A::m()`	`A::m()`
Scenario 17	*invokeinterface* `D::m()`	`D::m()`
	Message sent: `b.m()`	
Scenario 3	*invokeinterface* `A::m()`	`A::m()`
Scenario 11a	*invokeinterface* `B::m()`	`A::m()`
Scenario 11b	*invokeinterface* `B::m()`	`A::m()`
Scenario 17	*invokeinterface* `D::m()`	`D::m()`
	Message sent: `c.super(A).m()`	
Scenario 3	*invokespecial* `B::m()`	`A::m()`
Scenario 11a	*invokespecial* `A::m()`	`A::m()`
Scenario 11b	*invokespecial* `A::m()`	`A::m()`
Scenario 17	*invokespecial* `A::m()`	`D::m()`
	Message sent: `c.super(B).m()`	
Scenario 3	*invokespecial* `B::m()`	`A::m()`
Scenario 11a	*invokespecial* `B::m()`	`A::m()`
Scenario 11b	*invokespecial* `B::m()`	`A::m()`
Scenario 17	*invokespecial* `B::m()`	`D::m()`

Table 5.3: Results of EMB lookup

`B` is always the target class for a multisuper call and thus the instruction *invokespecial* `B::m()`. It is easy to see that `B::m()` will be identified as the EMB (after the recompilation of `B`) by the search process and is executed as a result.

Scenarios 11a and 11b

Assume that `A` is recompiled to remove the implementation for `m()`. Before the recompilation, the VMT of `C` should contain a slot pointing to a clone of `A::m()` which overrides `B::m()` in the original context. After recompilation, this slot points to a clone of `B::m()`. It is not difficult to see that both *invokevirtual* and *invokeinterface* will execute `B::m()`.

For *invokespecial*, after recompilation, `B::m()` will be located as the EMB whether `A` or `B` is the original target class because `B` exists in the IMT of both `A` and `B`. Therefore, the correct method is dispatched for all three invocations after recompilation.

Scenario 17

If only `A` is recompiled to insert a new overriding implementation for `m()`, the slot in the VMT of `C` points to a clone of `A::m()` because of the *relaxed multiple code inheritance* scheme. Thus, `A::m()` is executed for both *invokevirtual* and *invokeinterface*.

Because of the method lookup mechanism for *invokespecial*, `A::m()` will be located as the EMB and executed for the instruction *invokespecial* `A::m()`. `D::m()` will be located as the EMB for *invokespecial* `B::m()`. If only `B` is recompiled to insert a new overriding implementation for `m()` instead, `B::m()` is executed for *invokespecial* `B::m()` and `D::m()` is executed for *invokespecial* `A::m()`.

If both `A` and `B` are recompiled without recompiling all others, an ambiguity exception is thrown because the conflict between `A::m()` and `B::m()` cannot be resolved when building the VMT for `C`, even under the *relaxed multiple code inheritance*.

Conclusion

The modified method lookup procedure is able to find the most specific method at runtime, even after the recompilation of the *supertypes* for each of the scenarios discussed above. The modified JVM in MCI-Java will throw an exception if the recompilation causes unresolvable conflict.

5.5 Summary

In this chapter, the modifications to the Java virtual machine (JVM) are discussed. The modifications required to support MCI-Java include:

1. VMTs are added to the `ClassClass` structure of *implementations* to support multiple code inheritance in MCI-Java. The use of the `Virtual Method Table` (VMT) for *classes* only in classic Java is extended in MCI-Java to be applied to *interfaces* with code. It is built for an *implementation* to handle method inheritance between *implementations*, and to facilitate the introduction of methods declared in an *implementation* into the inheritance mechanism in the *class* hierarchy.

2. The new multisuper call is represented using the method invocation bytecode *invokespecial* in MCI-Java, taking an immediate *superimplementation* as the target class for method lookup. When the bytecode *invokespecial* is used for a multisuper call, a local and minor change is made to the original lookup procedure to find the resolved

`Method Block` (RMB) and execution `Method Block` (EMB), so that the original procedure is redirected to follow that for searching an interface method.

In the chapter that follows, the design and implementation of MCI-Java will be evaluated against the criteria set out in Section 3.1 in terms of compatibility, correctness and run-time performance.

This page contains no text

Chapter 6

Empirical evaluation

In this chapter, several empirical tests demonstrate that the MCI-Java VM – a modified version of Sun's JVM for Java 2 SDK 1.2.2 – and the modified IBM Jikes Compiler are compatible with their original programs, yielding correct results, as expected, with no performance degradation when compared with the performance of the original programs.

Cutumisu [15] refactors the `java.io` package to take advantage of multiple code inheritance, with techniques to be discussed below. The `.class` files for the refactored classes are generated by compiling the code using the modified IBM Jikes compiler, and applied using the MCI-Java VM. Through the tests, it is found that *invokespecial* is better suited for a multisuper call than the bytecode *invokeinterface*.

The implementation of MCI-Java is evaluated against the criteria set out in Section 3.1, in terms of compatibility, correctness and run-time performance.

The new language features of MCI-Java are applied to the standard Java run-time library (the file *rt.jar*), to demonstrate how they can assist in software engineering. The effect of refactoring some classes in the package `java.io` is also discussed in this chapter.

As pointed out in Section 3.1, the original semantics of Java should not be changed in MCI-Java, and the semantics of the new features should not be in conflict with those of the existing features of classic Java. These criteria are evaluated in this chapter, in terms of compatibility and correctness.

Finally, the performance of the new VM at run-time is compared with that of the original VM in machines running the Linux operating system.

6.1 Compatibility of MCI-Java for unmodified Java programs

The demonstration package `Java2D` accompanying Sun's Java 2 SDK 1.2.2 [3] was executed using the VM of both the classic Java and MCI-Java. Both executions gave the

same output.

The `java` packages from the classic Java library were compiled using both the original and modified version of Jikes[1] and the original `javac` (from SUN JDK 1.2.2), using the standard class library without any refactoring. Then, the `.class` files so generated are compared. Use of the `diff` program helps to verify that both the original and modified versions of each compiler generated the same output.

It can therefore be concluded that the modified compiler and VM are compatible with the original unmodified versions.

6.2 Correctness of MCI-Java generated code

The test for compatibility mentioned in the previous section indicates that the modified Jikes and the VM of MCI-Java give correct results executing legacy code. To verify the correctness with the new semantics, the 17 scenarios of multiple code inheritance discussed in Appendix D are coded, compiled using the modified Jikes, and executed using the VM of MCI-Java. In each of the tests, the execution results are exactly the same as described in Appendix D.

The issue of recompilation as discussed in Section 5.4.7 is also tested. The scenarios described in that section are coded and compiled using the modified Jikes. The execution of the generated code, using MCI-Java VM, gives the expected result in all cases.

6.3 Refactoring legacy code

In MCI-Java, a new reference *type*, *implementation*, is available for programmers to use as a separate *code-type*. Instead of using one single *implementation-type*, they can now separate code from data layout declaration, making it easier to re-use common code.

It is also important to be able to refactor existing reference *types*, so that future programs using existing *classes* can deploy the new language features provided by MCI-Java. Several techniques are used by Cutumisu [15] to refactor legacy code, and they are described below.

6.3.1 Duplicate code promotion

The simplest scenario is when two implementations of a method contain exactly the same code. The duplicated code is promoted to a method declared in an *implementation*. The two *classes* holding the duplicated code are then declared to `utilize` this newly

[1] IBM Jikes 1.15 Compiler.

```
1  public class DataInputStream {
2    ...
3    public final static String readUTF(DataInput in) throws IOException {
4      ··· code to be shared ···
5    }
6    public final String readUTF() throws IOException {
7      return readUTF(this);
8    }
9  }
10
11 public class RandomAccessFile {
12   ...
13   public final String readUTF() throws IOException {
14     return DataInputStream.readUTF(this);
15   }
16 }
```

(a) Code before refactoring

```
1  public implementation InputCode {
2    ...
3    public final String readUTF() throws IOException {
4      ··· code to be shared ···
5    }
6  }
7
8  public class DataInputStream utilizes InputCode {
9    ...
10   public final static String readUTF(DataInput in) throws IOException {
11     return in.readUTF();
12   }
13 }
14
15 public class RandomAccessFile utilizes InputCode ...
16
```

(b) Code after refactoring

Figure 6.1: Example for static method promotion

created *implementation*. The promoted code is invoked whenever an instance of either one of the two *classes* is sent a message for the promoted method.

6.3.2 Static method promotion

When two *classes* belong to two different inheritance hierarchies, it is impossible for them to share code by virtue of inheritance relationship between the two *classes*. In this situation, a static method is used to contain the common code, so that the code can be accessed via a static method call, by instances of the two *classes*. Figure 6.1 shows an example of applying the static method promotion technique to refactor two *classes* in the `java.io` package.

In Figure 6.1(a), `DataInputStream` (line 1) and `RandomAccessFile` (line 11)

belong to two different inheritance hierarchies. Common code shared between them is kept in the static method `readUTF()` (line 3), which is a member of `DataInputStream`. Each of the two *classes* has an instance method `readUTF()` (lines 6 and 13) implemented by invoking the static method passing the receiver object itself as an argument to the invocation.

Instead of keeping the shared code in a static method in one of the involved *classes* (`DataInputStream` in this case), the common code is now promoted, as shown in Figure 6.1(b), to become a method in the newly created *implementation* `InputCode` (line 1) from which both *classes* `utilize` (lines 8 and 15) for multiple code inheritance. However, there is still a static method in `DataInputStream` (line 10), so that legacy code using the original static method can be executed. The original code in the static method is now replaced with a message (line 11) sent to the receiver object, which is passed as an argument to the static method. This will invoke the promoted instance method in the common *implementation*.

6.3.3 Prefix method promotion

Figure 6.2 shows an example of applying the prefix method promotion technique to the `java.io` package. This technique is useful when two method implementations have identical code, differing only in some variables whose values are results of a set of class-dependent computation. The computation is performed before the common code is executed.

In Figure 6.2(a), two *classes* - `DataInputStream` and `RandomAccessFile` (lines 1 and 21) - have their own implementations for the method `readInt()` (line 3 and 23). It can be seen that the two implementations have identical code except for the local variable `in` (lines 5 - 9) used in `DataInputStream`, and `this` (lines 25 - 29) used in `RandomAccessFile`. In `DataInputStream`, the variable `in` is the result of the computation in line 5.

In Figure 6.2(b), a new *interface* (`Source`) (line 1) is created to serve as the common *interface* for the two involved *classes*, `RandomAccessFile` and `DataInputStream`, as well as the newly created *implementation* – `InputCode`. A method (`source()`) is declared in `Source` (line 2) for the *class*-dependent computation. Each *class* then implements this interface method using its unique code (lines 8 and 16). The common code, together with the *class*-dependent computation (embedded in the implementation of the interface method, lines 27 - 30), is then promoted to become the method `readInt()` (line

```
1  public class DataInputStream {
2    ...
3    public final int readInt()
4        throws IOException {
5      InputStream in = this.in;
6      int ch1 = in.read();
7      int ch2 = in.read();
8      int ch3 = in.read();
9      int ch4 = in.read();
10     if ( (ch1|ch2|ch3|ch4) < 0 )
11       throw new EOFException();
12
13     return (ch1 << 24) +
14            (ch2 << 16) +
15            (ch3 << 8) +
16            (ch4 << 0);
17   }
18 }
```

```
21 public class RandomAccessFile {
22   ...
23   public final int readInt()
24       throws IOException {
25
26     int ch1 = this.read();
27     int ch2 = this.read();
28     int ch3 = this.read();
29     int ch4 = this.read();
30     if ( (ch1|ch2|ch3|ch4) < 0 )
31       throw new EOFException();
32
33     return (ch1 << 24) +
34            (ch2 << 16) +
35            (ch3 << 8) +
36            (ch4 << 0);
37   }
38 }
```

(a) Code before refactoring

```
1  public interface Source {
2    public Source source();
3  }
4  public class DataInputStream
5               implements Source
6               utilizes InputCode {
7    ...
8    public Source source() {
9      return this.in;
10   }
11 }
12 public class RandomAccessFile
13              implements Source
14              utilizes InputCode {
15   ...
16   public Source source() {
17     return this;
18   }
19 }
```

```
21 public implementation InputCode
22              implements Source {
23   ...
24   public final int readInt()
25       throws IOException {
26
27     int ch1 = this.source().read();
28     int ch2 = this.source().read();
29     int ch3 = this.source().read();
30     int ch4 = this.source().read();
31     if ( (ch1|ch2|ch3|ch4) < 0 )
32       throw new EOFException();
33
34     return (ch1 << 24) +
35            (ch2 << 16) +
36            (ch3 << 8) +
37            (ch4 << 0);
38   }
39 }
```

(b) Code after refactoring

Figure 6.2: Example for prefix method promotion

24) in `InputCode`.

6.3.4 Super-suffix method promotion

Figure 6.3 depicts an example in which the super-suffix method promotion technique is deployed. In Figure 6.3(a), `DataOutputStream` and `RandomAccessFile` (lines 1 and 21) have essentially similar implementations for the same method - `writeInt()` (lines 3 and 23). The two implementations differ only in some epilogue statements (lines 10 and 30). The use of an additional local variable `out` in `DataInputStream` can be

```
1  public class DataOutputStream {
2    ...
3    public final void writeInt(int v)
4                throws IOException {
5      OutputStream out = this.out;
6      out.write((v >>> 24) & 0xFF);
7      out.write((v >>> 16) & 0xFF);
8      out.write((v >>> 8) & 0xFF);
9      out.write((v >>> 0) & 0xFF);
10     incCount(4);
11   }
12 }
```

```
21 public class RandomAccessFile {
22   ...
23   public final void writeInt(int v)
24               throws IOException {
25
26     write((v >>> 24) & 0xFF);
27     write((v >>> 16) & 0xFF);
28     write((v >>> 8) & 0xFF);
29     write((v >>> 0) & 0xFF);
30
31   }
32 }
```

(a) Code before refactoring

```
1  public class DataOutputStream implements Sink utilizes OutputCode {
2    ...
3    public Sink sink() {
4      return this.out;
5    }
6    public final void writeInt(int v) throws IOException {
7      super(OutputCode).writeInt(int v);
8      incCount(4);
9    }
10 }
11
12 public class RandomAccessFile implements Sink utilizes OutputCode {
13   ...
14   public Sink sink() {
15     return this;
16   }
17 }
18
19 public interface Sink {
20   public Sink sink();
21 }
22
23 public implementation OutputCode {
24   ...
25   public final void writeInt(int v) throws IOException {
26     Sink out = this.sink();
27     out.write((v >>> 24) & 0xFF);
28     out.write((v >>> 16) & 0xFF);
29     out.write((v >>> 8) & 0xFF);
30     out.write((v >>> 0) & 0xFF);
31   }
32 }
```

(b) Code after refactoring

Figure 6.3: Example for super-suffix method promotion

handled using the prefix method promotion technique described in the previous section.

In Figure 6.3(b), `OutputCode` (line 23) is a newly created common *implementation*. The common code in Figure 6.3(a) is factored out as the body of `writeInt()` (line 25) - declared in `OutputCode`. The epilogue statements stay in the method body of the original

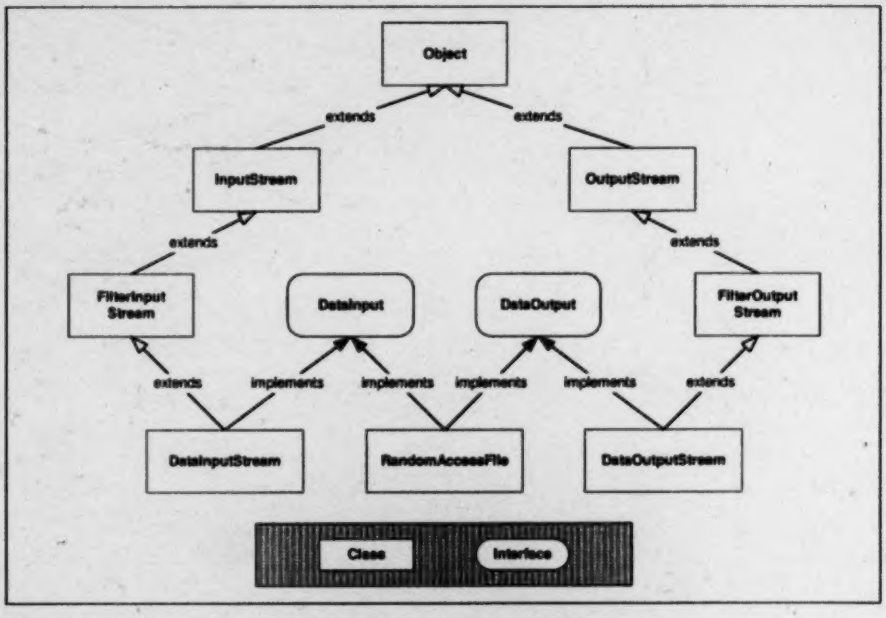


Figure 6.4: Hierarchy of the *classes* for data input/output before refactoring

instance method declared in the respective *classes* (line 8). The common code is invoked using a multisuper call before the epilogue statements are executed (line 7). `Sink` (line 19), which acts as a common *interface*, is introduced to define the method `sink()` (line 20) for the *class* dependent computation, as a part of the prefix method promotion scheme to handle the local variable `out` in `DataInputStream`.

6.4 Refactoring the `java.io` package

Figure 6.4 depicts the inheritance hierarchy of the original `java.io` package, showing only the *classes* involved with data input and output. The *classes* are now refactored (the hierarchy of the refactored *classes* is shown in Figure 6.6), using the techniques discussed in the previous section – resulting in a significant reduction in the total code size and the total number of declared methods. The most important impact of this refactoring process is that the refactored code is easier to understand, trace and maintain.

The *interface* `DataInput` has 15 declared abstract methods. These methods are implemented in two *classes* - `DataInputStream` and `RandomAccessFile`. Among them, 13 methods have similar implementations in both *classes*. Several of the techniques discussed in the previous section can be used to move common code into one location for

Duplicate code promotion	Prefix method promotion

`readFully(byte[],int,int)`
`readFully(byte[])`
`readFloat()`
`readDouble()`
	`readByte()`
	`readUnsignedByte()`
	`readShort()`
	`readUnsignedShort()`
	`readInt()`
	`readBoolean()`
	`readLong()`
	`readChar()`

Static method promotion	
`readUTF()`	

Table 6.1: Common code grouping by promotion technique for data input methods in `DataInputStream` and `RandomAccessFile`

Duplicate code promotion	`writeFloat()` `writeDouble()`

Super-suffix method promotion	`write(byte[],int,int)`
	`write(byte[])`
	`writeBoolean()`
	`writeByte()`
	`writeShort()`
	`writeInt()`
	`writeLong()`
	`writeChar()`

Table 6.2: Common code grouping by promotion technique for data output methods in `DataOutputStream` and `RandomAccessFile`

sharing by both *classes*. The methods can be grouped according to the applicable promotion techniques, as discussed in the previous section and shown in Table 6.1.

Similarly, the *interface* `DataOutput` has 14 declared abstract methods, which are implemented by `DataOutputStream` and `RandomAccessFile`. Among them, 10 methods have similar implementations in both *classes*. According to the applicable promotion techniques, these methods can be grouped as shown in Table 6.2.

As discussed in Section 6.3, two *implementations* (`InputCode` and `OutputCode`) are created to hold the promoted common code, while two newly created *interfaces* (`Sink` and `Source`) provide the common interface for generalization. Figure 6.5 shows the modified declaration lines for all the *classes*, *implementations* and *interfaces* involved. The hierarchy tree of the refactored package is shown in Figure 6.6. The *classes* and *interfaces* provided in the standard class library *rt.jar* are replaced by the refactored ones – including the newly created *implementations* and *interfaces* – to create a new run-time class library,

```
newly created types:
   public interface Source
   public interface Sink
   public implementation InputCode implements DataInput, Source
   public implementation OutputCode implements DataOutput, Sink
modified types:
   public abstract class InputStream implements Source
   public class FilterInputStream extends InputStream
   public class DataInputStream extends FilterInputStream utilizes InputCode
   public abstract class OutputStream implements Sink
   public class FilterOutputStream extends OutputStream
   public class DataInputStream extends FilterOutputStream utilizes OutputCode
   public class RandomAccessFile utilizes InputCode, OutputCode
```

Figure 6.5: Modified declarations for data input/output *types* in the `java.io` package

miRt.jar, for use in MCI-Java. Source code for the affected *interfaces*, *implementations* and *classes* are included in Appendix E.

The effect of refactoring the `java.io` package in reducing the number of methods used as well as the code size, has been tabulated below in Tables 6.3 and 6.4 (with abbreviated class names). The tables are based on the raw data listed in Appendix E. The meaning of the abbreviations are:

1. `RAF` stands for `RandomAccessFile`
2. `DIS` stands for `DataInputStream`
3. `DOS` stands for `DataOutputStream`
4. `IC` stands for `InputCode`
5. `OC` stands for `OutputCode`

The apparent low reduction rate in the number of methods used in `DataOutputStream` (as reflected in Table 6.3) is due to the fact that 6 of the 8 promotable methods contain an extra line of code unique to this *class*, making it necessary to retain a method declaration for the suffix. However the number of methods declared in the other two *classes*, namely `DataInputStream` and `RandomAccessFile`, are very significantly reduced.

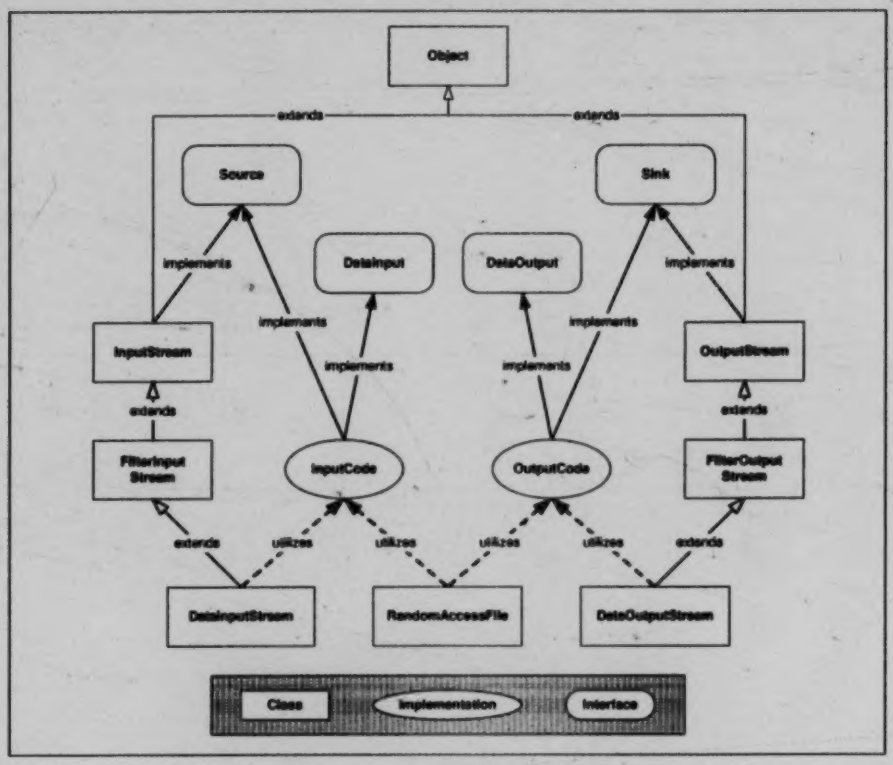


Figure 6.6: Hierarchy of the *classes* for data input/output

Class	RAF		DIS		DOS	
Method type	Before	After	Before	After	Before	After
Constructors	3	3	1	1	1	1
Native methods	11	11	0	0	0	0
Non-native methods	31	10	18	6[a]	16	15[b]
Total	45	24	19	7	17	16
Percentage reduction in methods	46.67%		63.16%		5.88%	

[a] a 1-line static method is kept for compatibility purposes
[b] 6 of the methods contain only 2 lines of code after super-suffix method promotion

Table 6.3: Change in the number of methods as a result of refactoring

It can also be seen from Table 6.4 that there is a significant reduction in the code size in each of the involved *classes*, and in the overall total.

Class	RAF	DIS	DOS	IC	OC	Overall
Lines not involved in the methods	93	42	54	-	-	189
Total lines before refactoring	158	127	84	-	-	369
Total lines after refactoring	93	44	67	82	27	313
Percentage Reduction in lines	41.14	65.35	25.37	-	-	15.18

Table 6.4: Change in the number of lines of code as a result of refactoring

6.5 Performance of Jikes and MCI-Java VM

One of the concerns about MCI-Java is the degradation in performance caused by the modifications performed on Jikes and the Java VM. The details of the two tests are provided in the following sections. Test results show that the modifications have negligible effect on the performance of both the compiler and the Java VM.

6.5.1 Compiler test

The original source files for the entire `java` package are used as the source files. They are compiled using the following compilers:

1. `javac` running in classic Java
2. `javac` running in MCI-Java
3. the unmodified Jikes
4. the modified Jikes

All four compilers use the original run-time library `rt.jar` as the bootclass library for compilation. For each compilation, the output is stored in a separate location and compared using the system program `diff`. As the output from Jikes and `javac` are different, `diff` is only applied to compare output from the same family of compilers. The result (stated in the previous section) is that there is no difference in output between the original and modified versions. A total of 100 runs is taken for each compiler in a Linux machine using AMD Athlon XP1800+ 1.5GHz CPU with 256KB cache. The 400 compilations are done in a continuous mode with each compiler taking turns so that the risk of performance being affected by background processes is spread evenly among them. The process times were reported using the system function `time`, and the results are tabulated in Table 6.5.

The small values in the standard deviation column indicate that the process times are very consistent between runs. Comparing the mean times for `javac` running in classic Java

	javac				jikes			
	classic Java		MCI-Java		original		modified	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev
real	64.581	0.474	63.645	0.463	3.082	0.085	3.085	0.107
user	61.990	0.418	61.024	0.180	1.769	0.075	1.772	0.075
sys	0.513	0.071	0.488	0.066	0.247	0.040	0.247	0.042

Unit of measure: seconds

Table 6.5: Compiler performance comparison

and MCI-Java (64.581 vs 63.645 and 61.990 vs 61.024) show that there is no significant difference in performance between the two versions of `javac`. However, the lower mean time for `javac` in MCI-Java is not an indication that it is performing better than the one in classic Java, because the diferences are within the standard deviation ranges. The mean times for Jikes (3.082 vs 3.085 and 1.769 vs 1.772) show that the performances of the two versions of Jikes are basically identical. It can be concluded that the modifications made to Jikes and the Java VM have negligible effect on the performance of both the compilers and the Java VM.

6.5.2 I/O test using refactored I/O *classes*

This test is primarily designed to test the refactored `java.io` package. The test program begins by writing a series of double precision values, integer values, character values and string values to an instance of `DataOutputStream`. The process is repeated 50,000 times. The data file is then closed and reopened as an instance of `DataInputStream`. The data are read from the file, line by line, and written to another file, opened as an instance of a `RandomAccessFile`. After the entire file is completely "copied", data are read from the instance of `RandomAccessFile` with the numerical values used to compute a sum for the correctness check and the correctness is verified. All the refactored methods are invoked during the program execution.

The test is performed on the same machine that is used for the compiler test. The program is run in the VM of both classic Java and MCI-Java 200 times, consecutively and alternately, in order to distribute the risks of background processes affecting the process times. The process times are recorded using the system function `time`. The results are tabulated in Table 6.6. When the test program is run on the classic VM, it uses the original `java.io` package. When it is run in MCI-Java, it uses the refactored `java.io` package that takes advantage of multiple code inheritance.

	classic JVM		MCI-Java VM	
	mean time	std deviation	mean time	std deviation
real	83.7807	2.6350	83.0003	2.4774
user	37.5243	0.5530	37.2336	0.5196
sys	45.9405	2.3724	45.4318	2.2410

Unit of measure: seconds

Table 6.6: VM performance comparison

The greater variations in performance, as reflected in the value of the standard deviation for real time (2.6350 and 2.4774), are mainly due to the I/O component of the tests. It can be seen that the process times for the non I/O part are very consistent (standard deviations 0.5530 and 0.5196). Despite the greater variations in the overall process times, they demonstrate similar distribution patterns with very little differences in the mean and standard deviation values. The mean real times are 83.7807 and 83.003, with standard deviations 2.6350 and 2.4774. The mean user times are 37.5243 and 37.2336 with standard deviations 0.5530 and 0.5196. The numbers show that there is no significant difference between the performance of the classic JVM and MCI-Java VM. Based on this observation, it can be concluded that the modifications made to the Java VM and the refactored classes have no significant effect on the overall performance.

6.6 Summary

It has been demonstrated in this chapter that legacy code can be refactored to take advantage of the new features introduced in MCI-Java, using the following four method promotion techniques:

1. Duplicate code promotion
2. Static method promotion
3. Prefix method promotion
4. Super-suffix method promotion

The techniques are applied to the original `java.io` package. Refactoring has successfully reduced both the total code size and the combined number of methods used.

Through the various tests conducted, it is shown that the MCI-Java VM, the modified version of Sun's JVM for Java 2 SDK 1.2.2 and the modified IBM Jikes Compiler are

compatible with the original programs, give correct results as expected and – most important of all – do not degrade the performance when compared with the original programs.

Chapter 7

Conclusions

The design and implementation of MCI-Java, an extended Java Virtual Machine supporting multiple code inheritance, has been presented in this dissertation. The modifications made to the IBM Jikes Compiler [4] to support the newly introduced syntax in MCI-Java are also presented. This dissertation is concluded with a discussion on the related work of other researchers, a summary of the work done, future directions and research contributions.

7.1 Related work

Many researchers have been working on schemes that allow multiple code inheritance in programming languages originally designed with single inheritance. Some of their work is discussed below.

7.1.1 Mixins

A mixin is a specification with methods. It can be used to extend various classes with the same set of features.

Mixin-based inheritance, proposed by Bracha and Cook [9], sees a mixin as the primary construct, while inheritance is built from mixins. This concept was applied to Modula-3.

Flatt *et el.* [17] proposed application of the concept of mixins to Java. In MIXEDJAVA, a mixin is viewed as a function that extends a class by adding methods. An atomic mixin in MIXEDJAVA declares a new mixin, while a composite mixin composes two existing mixins to create a new one. This idea has not yet been implemented in any version of Java.

Ancona *et el.* [7] extends Java 1.0 with mixins. The translator `jamc` translates JAM code into Java code by mapping a mixin declaration into an interface. Every time a new class is required, `jamc` is invoked before `javac`, thus making it a very expensive process. Furthermore, the variable `this` is forbidden in mixins in JAM.

There is an excellent overview of mixins in [29] which illustrates the fact that mixin inheritance [9] does not work well when more than one mixin is used by a *class*, because "the mixins do not quite fit together" [29]. Three main problems with mixins have been identified, as described below:

1. Linearization - Mixin-based inheritance, proposed by Bracha and Cook [9], requires explicit linearization of the mixins. A *class* is viewed as a composition of mixins. When attributes are in conflict, the one from the newly added mixin will override the older ones. However, a suitable total order does not always exist for conflict removal. The problem of total ordering does not exist in MCI-Java, as inheritance is symmetrical.

2. Mixin programming - Special code (glue code) is written to link mixins together. Glue code reduces flexibility and also gives rise to maintenance issues. In MCI-Java, glue code is not needed and therefore the associated problems do not exist.

3. Fragile hierarchies - A newly added method may have the unwanted side effect of silently overriding another method, thus changing some behaviours because of the required total ordering. Since conflict removal is hard coded, using glue code, a change to a mixin must be rippled across the inheritance hierarchy when a mixin is used in several places. Most of the problems are solved in MCI-Java with the multiSuperCall mechanism and the adoption of the *relaxed multiple code inheritance* approach for conflict resolution. Most of the errors are reported at load time during the VMT building process, and any errors that cannot be determined at load time are handled as exceptions at run-time, thereby ensuring that there are no unexpected executions.

7.1.2 Traits

Schärli *et el.* [29] implemented *traits* as a solution to multiple code inheritance in the Squeak dialect of Smalltalk [20].

Traits are similar to *implementations* proposed in this dissertation in the following areas:

1. Each is a collection of methods, possibly invoking methods that are abstract until they are natively defined in a client class;

2. Each can be used by a client class to augment its natively-defined methods;

3. Each contains no representation information;

4. A class can use more than one *trait* or *implementation*.

However, the two solutions to multiple code inheritance have the following major differences:

1. When a client class that uses a *trait* is compiled, the non-overridden methods are flattened into the client class by extending the method dictionary of the client class with one entry for each non-overridden method. The entries in all method dictionaries using the same *trait* point to a common code stored in the *trait* which is a "hidden class". An *implementation* is a first-class language feature which is not hidden and has its own VMT for method lookup.

2. When a *trait* or a superclass is recompiled, the client class is automatically recompiled. This is made possible by the fact that all source codes are stored in the same Smalltalk image. However, automatic recompilation of the client class is not possible and is not required in Java and MCI-Java.

3. While both *trait* and *implementation* solve the "diamond" inheritance problem by declaring no conflict, they handle potential conflict differently. Some are classified as conflicting scenarios by *traits*, but not with *implementations*, because of the adoption of the *relaxed multiple code inheritance* approach. They also select the method to be invoked differently. Methods from *traits* take precedence over those from superclasses, but *implementations* treat methods from all sources the same.

4. The inheritance semantics used in *trait* relies on compile-time based flattening technique, which is difficult to support in Java because of the inaccessibility of source code at load time. *Implementations* follows the dynamic semantics of Java.

7.1.3 Code in *interface*

Cutumisu[14] has implemented code in interface as a solution to multiple code inheritance in Sun JDK 1.2.2. This implementation differs from the work presented in this dissertation in the following areas:

1. Cutumisu's implementation used the Miranda Method[1] technique to insert a method declared in *interface* into the VMT of an inheriting *class*. No VMT is built for an *interface*. MCI-Java has VMT for *implementations* - *interfaces* with code.

[1] In Sun's JVM, an abstract method declared in an abstract superclass is classified as the Miranda Method, and is reserved a slot in the VMT for future implementation defined in one of the subclasses

2. A special scripting process is designed to transform specially written program files for *interfaces* containing code into binary format `.class` files.

3. The modified JVM has to be operated in the "-noverify" mode, as the code verifier at load time is not adjusted to accommodate code in interface.

7.2 Summary of work

This dissertation starts with a scenario illustrating the need for multiple code inheritance and separated *interface-type*, *code-type* and *data-type*. This improves expressiveness and allows programmers more code re-use, avoiding duplicated code and resulting in fewer maintenance problems.

Subsequently, a brief review of the current Java technology, with the emphasis on its weaknesses in handling multiple representations and multiple behavior specifications, is given. The difficulty in sharing code across inheritance hierarchy trees is also highlighted.

MCI-Java is proposed as a solution. MCI-Java supports the total separation of *interface-type*, *code-type* and *data-type* with the introduction of, at the source code level, *implementation* - a new type declaring unit, and `utilizes` - a new keyword in the Java syntax signifying inheritance from an *implementation* by a *class*. The semantics of the new construct are explained in detail, with 17 scenarios illustrated for elaboration. As a side effect of the language extension, the multiSuper call mechanism is also introduced, with its semantics fully explained.

To support the new language syntax, the IBM Jikes Compiler is modified. A detailed explanation of the modification follows a description of the existing operation of the original compiler. Prior to describing the modifications made to Sun's JVM by MCI-Java, the operation of those parts of the JVM involved in method dispatch are explained.

Several experiments are conducted to verify that the modified compiler behaves correctly in handling code written using the classic Java semantics, as well as the semantics of MCI-Java. It is also verified that the modified JVM executes code correctly, according to the defined semantics. No measurable difference in performance is found between the original and the modified compiler and VM.

7.3 Research contribution

The research contributions of this dissertation include:

1. The first implementation with compiler support to extend Java to support the separation of *interface-type*, *code-type* and *data-type*, with a VM that supports code generated by the modified compiler. The modified JVM is able to support verification of code in *interfaces* without affecting the original verification processes.

2. Language syntax changes are decoupled from the modifications in the VM. New syntax is introduced in the source code level only. The modified JVM can be used to execute any code produced by any compiler that supports code in *interfaces*. Similarly, the modified IBM Jikes compiler can be used to support any extensions to Java requiring three separate *type* declaring units.

3. A super call mechanism similar to the one found in C++ is implemented with compiler support. This mechanism allows programmers to specify an inheritance path to the desired method implementation. Furthermore, this mechanism supports the dynamic Java semantics that permits recompilation of any one of the intermediate *classes*, and is still able to locate the most specific implementation for invocation at run-time.

4. An empirical evaluation of the MCI-Java VM shows that there is no measurable difference in performance between the original JVM and the MCI-Java JVM.

The most important modification to the JVM is the construction of the virtual method table (VMT) for an `interface`. An `implementation`, the newly introduced language construct at the source code level, is mapped to an *interface* at the VM level. The construction of the VMT facilitates the identification of method implementation that is visible at any point of an inheritance tree. It also helps to identify potential inheritance conflict at load time.

All changes to the JVM and the Jikes compiler are localized. In the original JVM all method-invoking bytecodes are quick-ed and then executed using the highly optimized module written in assembly language. The module has not been changed and the algorithm for executing the new *multiSuperCall* is designed in such a way that the same module for quick-ed bytecode execution can be used.

This thesis has proposed and implemented a scheme to improve the expressiveness of the original Java platform. MCI-Java facilitates code re-use, supports the separation of inheritance, and allows programmers to write less code without affecting the clarity of their implementations. Existing program code and class libraries are not affected by

the modifications, and there is no measurable effect on performance. The modified Jikes Compiler has been verified to generate correct `.class` files for both traditional programs and programs using the proposed new features. Execution of the generated code in the modified JVM is shown to be correct for the programs using classic Java semantics, as well as those using the new multiple code inheritance and super call mechanisms.

Bibliography

[1] http://java.sun.com/docs/books/tutorial/collections/implementations/general.html. webpage. Last accessed July 23, 2003.

[2] http://java.sun.com/j2se/1.4.2/docs/api/index.html. webpage. Last accessed November 17, 2003.

[3] http://java.sun.com/software/communitysource/j2se/java2/download.html. webpage. Last accessed November 17, 2003.

[4] http://www.ibm.com/developerworks/opensource/jikes/. webpage. Last accessed December 22, 2003.

[5] http://www.jaggersoft.com/csharp_standard/. webpage. Last accessed July 21, 2003.

[6] http://www.opensource.org/. webpage. Last accessed December 22, 2003.

[7] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam — A smooth extension of Java with mixins. *Lecture Notes in Computer Science*, 1850:145–178, 2000.

[8] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *SIGPLAN Not.*, 23(SI):1–142, 1988.

[9] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *OOPSLA/ECOOP '90: Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[10] Timothy Budd. *An Introduction to Object-Oriented Programming, Second edition.* Addison Wesley Longman, Inc., 1997.

[11] Craig Chambers and Gary Leavens. The Cecil language: Specification and rationale. Technical Report TR93-03-05, Department of Computer Science, FR-35, University of Washington, 1993.

[12] Craig Chambers and Gary Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(6):805–843, 1995.

[13] Craig Chambers and Gary Leavens. BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing. In *The Fourth International Workshop on Foundations of Object-Oriented Languages, FOOL 4, Paris, France*, 1996.

[14] Maria Cutumisu. Multiple Code Inheritance in Java. Master's thesis, University of Alberta, 2003.

[15] Maria Cutumisu, Calvin Chan, Paul Lu, and Duane Szafron. MCI-Java: A Modified Java Virtual Machine Approach to Multiple Code Inheritance. In *3rd Virtual Machine Research and Technology Symposium Usenix VM '04 Conference Proceedings*, pages 13–28, San Jose, California, United States, May 2004.

[16] Christopher Dutchyn. Multi-dispatch in the Java virtual machine : Design, implementation, and evaluation. Master's thesis, University of Alberta, 2002.

[17] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

[18] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[19] M. Huchard. *Sur quelques questions algorithmiques de l'héritage multiple*. PhD thesis, Université Montpellier II, 1988.

[20] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 318–326. ACM Press, 1997.

[21] Yuri Leontiev, Tamer Özsu, and Duane Szafron. On Separation between Interface, Implementation, and Representation in Objects DBMSs. In *26th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA98)*, pages 155 – 167, Santa Barbara, August 1998.

[22] Jesse Liberty. *Object-Oriented Analysis and Design with C++*. Wrox Press Ltd, 1998.

[23] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1997.

[24] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, pages 17–34. ACM Press, 1987.

[25] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, 1974.

[26] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[27] Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. In *HOPL-1: The first ACM SIGPLAN Conference on History of Programming Languages*, pages 245–272. ACM Press, 1978.

[28] Candy Pang, Wade Holst, Yuri Leontiev, and Duane Szafron. Multi-Method Dispatch Using Multiple Row Displacement. In *ECOOP 1999 Conference Proceedings*, pages 304–328. 13th European Conference on Object-Oriented Programming, Lisbon, Portugal, June 1999.

[29] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP 2003 Conference Proceedings*, pages 248–274, Darmstadt, Germany, July 2003. 17th European Conference on Object-Oriented Programming.

[30] Alan Snyder. Inheritance and the development of encapsulated software systems. In *Research directions in object-oriented programming*, pages 165–188. MIT Press, 1987.

[31] David Stoutamirer and Stephen Omohundro. The Sather 1.0 Specification. Technical Report TR96-012, International Computer Science Institute of Berkeley, 1996.

[32] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

[33] Peter Wegner. Dimensions of object-based language design. In *OOPSLA '87: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 168–182. ACM Press, 1987.

This page contains no text

Appendix A

Abbreviations

Many abbreviations have been used in this thesis. Table A.1 is a collection of these abbreviations with their whole forms and meanings.

Abbreviation	Whole form	Meaning
EMB (§5.4.4)	Execution Method Block	the "in memory" data structure holding information about the method to be executed
EMT (§4.3)	Expanded Method Table	an "in memory" symbol table for methods used by Jikes to handle code inheritance
IMT (§5.3)	Interface Method Table	an "in memory" data structure used to identify the slot number for methods declared in an *interface*
JVM (§3.1)	Java Virtual Machine	an abstract computing machine in which a program written in Java runs
MT (§5.3)	Method Table	an "in memory" data structure used to hold information about all the methods declared a *type*
OOP (§1.1)	Object Oriented Programming	a programming paradigm in which a problem is solved using interactive message passing among objects
RCP (§5.1.1)	Run-time Constant Pool	a run-time symbol table inside the Java Virtual Machine
RMB (§5.4.2)	Resolved Method Block	the "in memory" data structure holding information about the method referred to by an entry in the run-time constant pool
VM (§3.3)	Virtual Machine	an abstract computing machine
VMT (§5.3)	Virtual Method Table	an "in memory" data structure used to identify the most specific methods visible in a *type*

Table A.1: Meaning of the abbreviations used

This page contains no text

Appendix B

Class File Structure

The new schematic diagram of the modified class file structure is shown in Figure B.1. A new class access flag (`IMPLEMENTATION`) has been added on top of the original five. As a result, the legitimate class access flags are:

1. `PUBLIC`
2. `FINAL`
3. `SUPER`
4. `ABSTRACT`
5. `INTERFACE`
6. `IMPLEMENTATION`

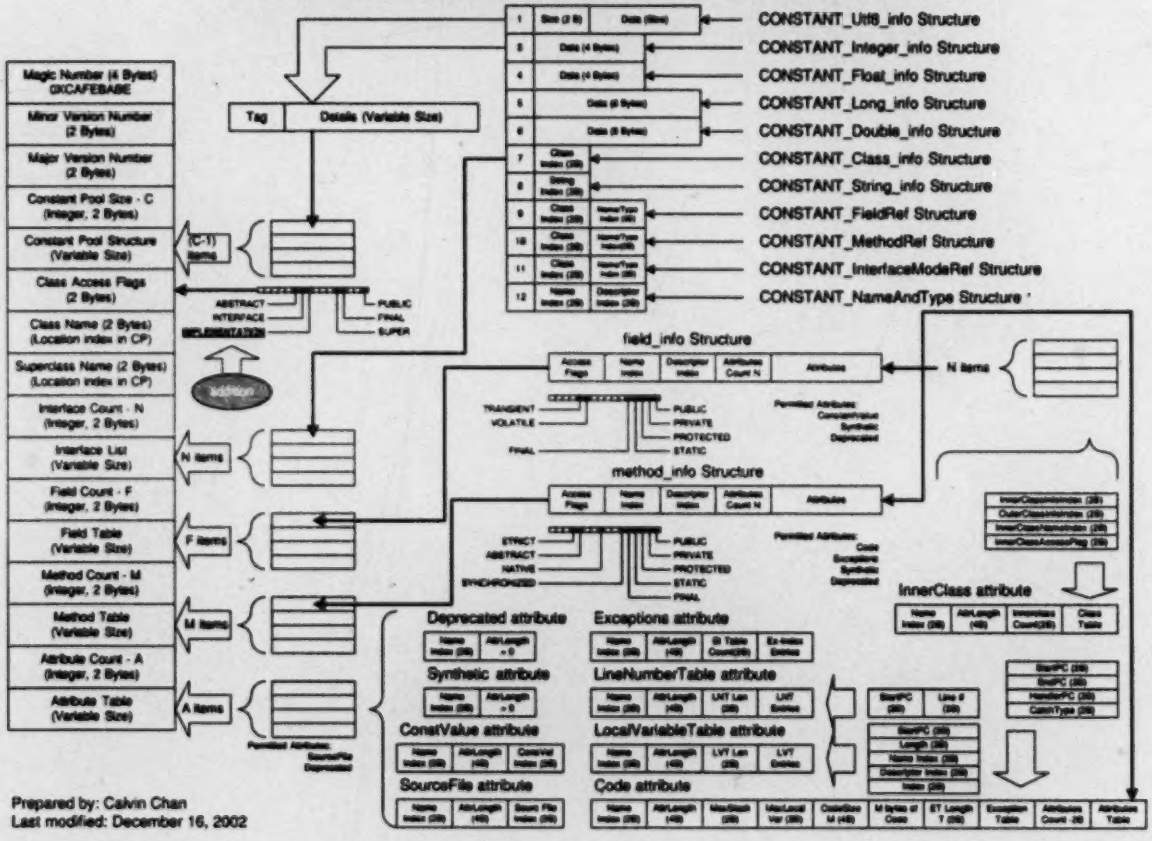


Figure B.1: Schematic Diagram of the Modified Class File Structure

Appendix C

The Grammar of MCI-Java

C.1 Introduction

This chapter presents a grammar for MCI-Java suited for a parser. This version is based on the Java grammar included in Chapter 18 of the book **The Java™ Language Specification, second edition** (JLS) by James Gosling, Bill Joy, Guy Steele, and Gilad Bracha [18].

The starting goal symbol is *CompilationUnit*. The production line has been marked with ➢➢ for easy identification.

Similar to the original Java grammar, the grammar below uses the following BNF-style conventions:

* *[x]* denotes zero or one occurrences of *x*.
* $\left\{x\right\}$ denotes zero or more occurrences of *x*.
* $\left(x \mid y\right)$ means one of either *x* or *y*.
* `keyword` is an actual keyword used in MCI-Java.

This grammar for MCI-Java, which is included in Section C.2, is generated by modifying the original Java grammar. Changes to the original version are recorded as

1. additions,
2. ~~removals~~ or
3. added new non-terminal:
   rules for the added non-terminal

C.2 The Grammar of MCI-Java

Identifier:
IDENTIFIER

QualifiedIdentifier:
Identifier { . *Identifier* }

Literal:
IntegerLiteral
FloatingPointLiteral
CharacterLiteral
StringLiteral
BooleanLiteral
NullLiteral

Expression:
Expression1 [*AssignmentOperator Expression1*]

AssignmentOperator:
=
+ =
− =
* =
/ =
& =
| =
^ =
% =
<<=
>>=
>>>=

Type:
Identifier { . *Identifier* } *BracketsOpt*
BasicType

StatementExpression:
Expression

ConstantExpression:
Expression

Expression1:
Expression2 [*Expression1Rest*]

Expression1Rest:
[? *Expression* : *Expression1*]

Expression2 :
Expression3 [*Expression2Rest*]

Expression2Rest:

{ *Infixop Expression3* }

Expression3 `instanceof` *Type*

Infixop:

`||`
`&&`
`|`
`^`
`&`
`==`
`!=`
`<`
`>`
`<=`
`>=`
`<<`
`>>`
`>>>`
`+`
`-`
`*`
`/`
`%`

Expression3:

PrefixOp Expression3

`(` (*Expr* | *Type*) `)` *Expression3*

Primary { *Selector* } { *PostfixOp* }

Primary:

`(` *Expression* `)`
`this` [*Arguments*]
`super` *SuperSuffix*
Literal
`new` *Creator*
Identifier { `.` *Identifier* } [*IdentifierSuffix*]
BasicType BracketsOpt `.class`
`void.class`

IdentifierSuffix:

`[` (`]` *BracketsOpt* `.class` | *Expression* `]`)
Arguments
`.` (`class` | `this` | `super` *Arguments* | `new` *InnerCreator*)

PrefixOp:

`++`
`--`

!
~
+
-

PostfixOp:
++
--

Selector:
. Identifier [Arguments]
`.this`
. `super` *SuperSuffix*
. `new` *InnerCreator*
[Expression]

SuperSuffix:
Arguments
[(Identifier)] . Identifier [Arguments]

BasicType:
`byte`
`short`
`char`
`int`
`long`
`float`
`double`
`boolean`

ArgumentsOpt:
[Arguments]

Arguments:
([Expression { , Expression }])

BracketsOpt:
{[]}

Creator:
QualifiedIdentifier (ArrayCreatorRest | ClassCreatorRest)

InnerCreator:
Identifier ClassCreatorRest

ArrayCreatorRest:
[( ] BracketsOpt ArrayInitializer | Expression] { [Expression] } BracketsOpt)

ClassCreatorRest:
Arguments [ClassBody]

ArrayInitializer:

{ [*VariableInitializer* { , *VariableInitializer* } [,]] }

VariableInitializer:
ArrayInitializer
Expression

ParExpression:
(*Expression*)

Block:
{ *BlockStatements* }

BlockStatements:
{ *BlockStatement* }

BlockStatement:
LocalVariableDeclarationStatement
ClassOrInterfaceDeclaration
[Identifier :] Statement

LocalVariableDeclarationStatement:
[final] Type VariableDeclarators ;

Statement:
Block
`if` *ParExpression Statement* [`else` *Statement*]
`for` (*ForInitOpt* ; [*Expression*] ; *ForUpdateOpt*) *Statement*
`while` *ParExpression Statement*
`do` *Statement* `while` *ParExpression* ;
`try` *Block* (*Catches* | [*Catches*] `finally` *Block*)
`switch` *ParExpression* { *SwitchBlockStatementGroups* }
`synchronized` *ParExpression Block*
`return` [*Expression*] ;
`throw` *Expression* ;
`break` [*Identifier*]
`continue` [*Identifier*]
;
ExpressionStatement
Identifier : Statement

`Originally missing in chapter 18, from section 14.8 of JLS`
ExpressionStatement:
StatementExpression ;

Catches:
CatchClause { *CatchClause* }

CatchClause:
`catch` (*FormalParameter*) *Block*

SwitchBlockStatementGroups:

{ *SwitchBlockStatementGroup* }

SwitchBlockStatementGroup:

SwitchLabel BlockStatements

SwitchLabel:

`case` *ConstantExpression* :

`default` :

MoreStatementExpressions:

{ , *StatementExpression* }

ForInit:

StatementExpression MoreStatementExpressions

[`final`] *Type VariableDeclarators*

ForUpdate:

StatementExpression MoreStatementExpressions

ModifiersOpt:

{ *Modifier* }

Modifier:

`public`
`protected`
`private`
`static`
`abstract`
`final`
`native`
`synchronized`
`transient`
`volatile`
`strictfp`

VariableDeclarators:

VariableDeclarator { , *VariableDeclarator* }

VariableDeclaratorsRest:

VariableDeclaratorRest { , *VariableDeclarator* }

ConstantDeclaratorsRest:

ConstantDeclaratorRest { , *ConstantDeclarator* }

VariableDeclarator:

Identifier VariableDeclaratorRest

ConstantDeclarator:
Identifier ConstantDeclaratorRest

VariableDeclaratorRest:
BracketsOpt [= VariableInitializer]

ConstantDeclaratorRest:
BracketsOpt = VariableInitializer

VariableDeclaratorId:
Identifier BracketsOpt

➢➢*CompilationUnit:*
[`package` *QualifiedIdentifier* ;] { *ImportDeclaration* } { *TypeDeclaration* }

ImportDeclaration:
`import` *Identifier* { . *Identifier* } [. *] ;

TypeDeclaration:
ClassOrInterfaceDeclaration
;

ClassOrInterfaceDeclaration:
ModifiersOpt (*ClassDeclaration* | *ImplementationDeclaration* | *InterfaceDeclaration*)

ClassDeclaration:
`class` *Identifier* [`extends` *Type*] [`implements` *TypeList*] [`utilizes` *TypeList*]
ClassBody

ImplementationDeclaration:
`implementation` *Identifier* [`extends` *TypeList*] [`implements` *TypeList*]
ImplementationBody

InterfaceDeclaration:
`interface` *Identifier* [`extends` *TypeList*] *InterfaceBody*

TypeList:
Type { , *Type* }

ClassBody:
{ { *ClassBodyDeclaration* } }

ImplementationBody:
{ { *ImplementationBodyDeclaration* } }

InterfaceBody:

{ { *InterfaceBodyDeclaration* } }

ClassBodyDeclaration:
;
[`static`] *Block*
ModifiersOpt MemberDecl

MemberDecl:
MethodOrFieldDecl
`void` *Identifier MethodDeclaratorRest*
Identifier ConstructorDeclaratorRest
ClassOrInterfaceDeclaration

MethodOrFieldDecl:
Type Identifier MethodOrFieldRest

MethodOrFieldRest:
VariableDeclaratorRest
MethodDeclaratorRest

InterfaceBodyDeclaration:
;
ModifiersOpt InterfaceMemberDecl

InterfaceMemberDecl:
InterfaceMethodOrFieldDecl
~~`void` *Identifier InterfaceMethodDeclaratorRest*~~
`void` *Identifier AbstractMethodDeclaratorRest*
ClassOrInterfaceDeclaration

InterfaceMethodOrFieldDecl:
Type Identifier InterfaceMethodOrFieldRest

InterfaceMethodOrFieldRest:
ConstantDeclaratorsRest ;
AbstractMethodDeclaratorRest
~~*InterfaceMethodDeclaratorRest*~~

ImplementationBodyDeclaration:
;
ModifiersOpt ImplementationMemberDecl

ImplementationMemberDecl:
Type Identifier ConstantDeclaratorsRest ;
Type Identifier MethodDeclaratorRest
`void` *Identifier VoidMethodDeclaratorRest*
ClassOrInterfaceDeclaration

MethodDeclaratorRest:
FormalParameters BracketsOpt [`throws` *QualifiedIdentifierList*] (*MethodBody* | ;)
AbstractMethodDeclaratorRest

AbstractMethodDeclaratorRest:
~~*InterfaceMethodDeclaratorRest:*~~
FormalParameters BracketsOpt [`throws` *QualifiedIdentifierList*] `;`

ConstructorDeclaratorRest:
FormalParameters [`throws` *QualifiedIdentifierList*] *MethodBody*

QualifiedIdentifierList:
QualifiedIdentifier { `,` *QualifiedIdentifier* }

FormalParameters:
`(` [*FormalParameter* { `,` *FormalParameter* }] `)`

FormalParameter:
[`final`] *Type VariableDeclaratorId*

MethodBody:
Block

This page contains no text

Appendix D

Multiple Code Inheritance Scenarios

Seventeen multiple code inheritance scenarios, which are commonly found in normal application programming, are described in this chapter. The outcome of a method invocation in each of the scenarios is examined using the semantics of the extended language.

The legend used in the diagrams for depicting the scenarios is shown in Figure D.1. A *class* and an *implementation* are represented by a rectangular and an oval shape, respectively. A single alphabet in capital letters denotes the name of the *type*.

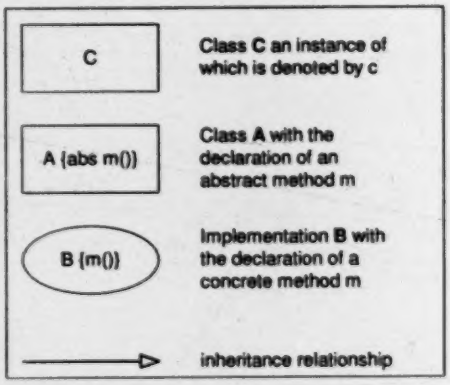

C
Class C an instance of which is denoted by c
A {abs m()}
Class A with the declaration of an abstract method m
B {m()}
Implementation B with the declaration of a concrete method m
inheritance relationship

Figure D.1: Legend for scenario diagrams

When the *type* in question contains a definition for a method, the single alphabet will be followed by the name of the method enclosed in a pair of curly brackets. An abstract method declaration is identified with the three letters 'abs'. It should also be noted that a method definition contained in a *type* may have the meaning extended to include cases where the method is inherited from the *supertypes*, and is visible at that point in the inheritance hierarchy. To simplify the explanation, `c` is used to denote a run-time instance of `C`, and `c.m()` denotes a message sent to this instance of `C`.

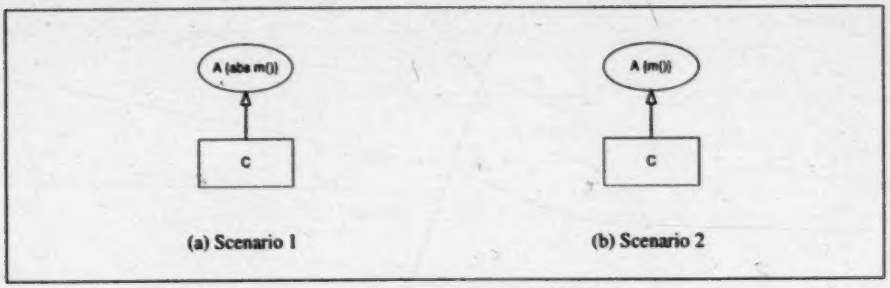


Figure D.2: Scenarios 1 - 2

D.1 Single inheritance involving an *implementation*

D.1.1 Scenarios 1 - 2

Figure D.2(a) shows a situation in which *class* C `utilizes` *implementation* A. An abstract method `m()` is declared in A. At compile time, the compiler will convert C into an abstract *class* because there is an abstract method visible in C, but no concrete implementation of `m()` is found in C. If the situation is a result of a recompilation of A without recompiling C, a run-time exception will be raised when the message `c.m()` is sent.

However, in Scenario 2 (as shown in Figure D.2(b)), *implementation* A contains the concrete method `m()` instead. This method is therefore visible in C by virtue of the inheritance relationship between A and C. At run-time, the message `c.m()` will be dispatched to the method found in A unless it is overridden in C by another implementation of the method `m()`, according to the semantics of the extended language.

D.1.2 Scenario 3

In Scenario 3 (as shown in Figure D.3(a)), *class* C `utilizes` *implementation* B which, in turn, `extends` another *implementation* A. While the implementation for method `m` is found in A, the method is not overridden in B and C. Thus, the method definition found in A is the one visible in C. The message `c:m()` at run-time is dispatched to the method found in A. The outcome is similar to the normal inheritance scenarios seen with *classes*.

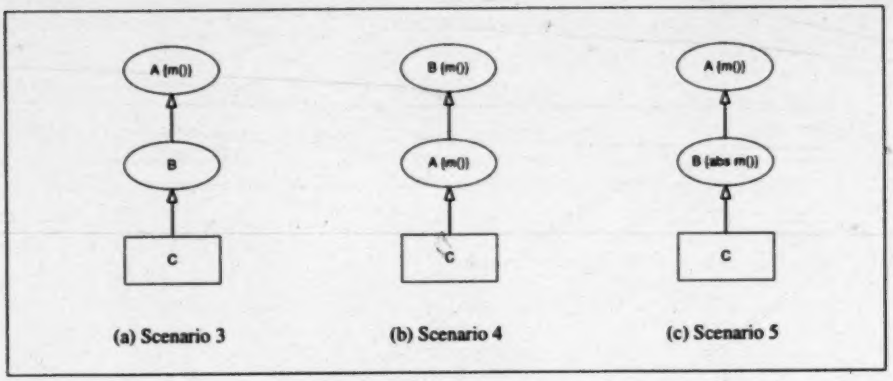


Figure D.3: Scenarios 3 - 5

D.1.3 Scenario 4

Scenario 4, depicted in Figure D.3(b), is similar in inheritance relationship between *types* to Scenario 3, discussed in the previous section. However, the implementation for the method `m()` found in `B` is overridden by the one found in `A`, and is therefore no longer visible in the inheritance hierarchy in the subtree rooted from `A`.

In `A` and `C`, only the implementation defined in `A` is visible and thus the message `c.m()` will be dispatched to the method found in `A`, according to the semantics of the extended language.

D.1.4 Scenario 5

Scenario 5, shown in Figure D.3(c), is a special case. Section 8.4.3.1 of the Java Language Specification [18] states that "An instance method that is not `abstract` can be overridden by an `abstract` method."

The phenomenon, known as inheritance suspension, is intended to force a new implementation for the method being overridden. Thus, the definition of `m()` found in `A`, being suspended by the abstract method in `B`, is no longer available in `C`. This scenario is therefore equivalent to the situation found in Scenario 1. When a message `c.m()` is sent at run-time, an exception will be raised – the same result as for Scenario 1.

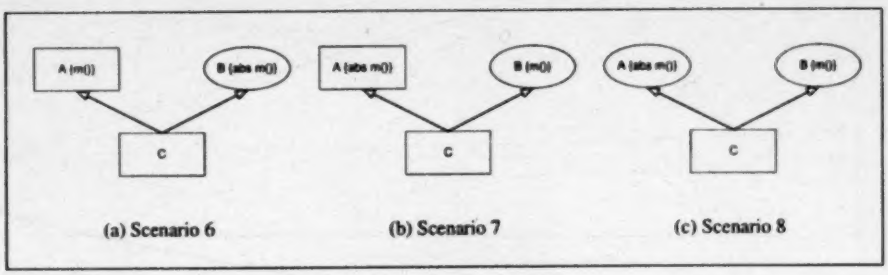


Figure D.4: Scenarios 6 - 8

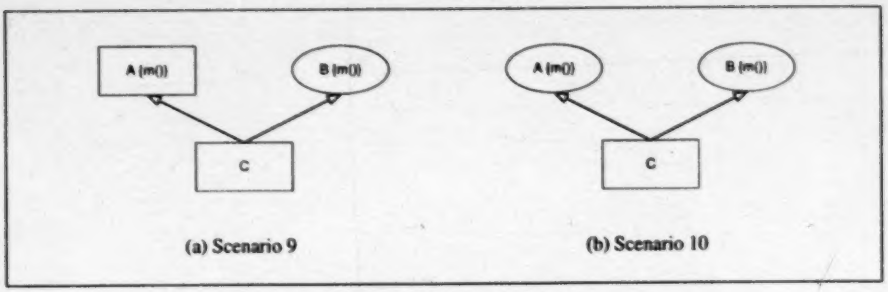
D.2 Multiple inheritance involving an *implementation*

D.2.1 Scenario 6

The class hierarchy is depicted in Figure D.4(a). In this scenario, `C extends` ***class*** `A` and `utilizes` ***implementation*** `B`. `A` contains a definition for the concrete method `m()`, while `B` declares an abstract method with the same signature. It is a conflict situation because two distinct declarations with the same signature are visible in `C`. According to the semantics of the extended language, concrete method is given precedence over abstract method in resolving conflicts and thus only the definition from `A` is made visible in `C`. At run-time, the message `c.m()` is dispatched to the method in `A`.

D.2.2 Scenario 7

The class hierarchy is depicted in Figure D.4(b). In this scenario, `C extends` ***class*** `A` and `utilizes` ***implementation*** `B`. `B` contains a definition for the concrete method `m()`, while `A` declares an abstract method with the same signature. It is a conflict situation because two distinct declarations with the same signature are visible in `C`. According to the semantics of the extended language, concrete method is given precedence over abstract method in resolving conflicts and thus only the definition from `B` is made visible in `C`. At run-time, the message `c.m()` is dispatched to the method in `B`. The outcome at run-time is similar to that for Scenario 6, except that the roles of `A` and `B` are swapped in both cases. This demonstrates the fact that no precedence is given to either a *class* or an *implementation* in resolving conflicts.



(a) Scenario 9 (b) Scenario 10

Figure D.5: Scenarios 9 - 10

D.2.3 Scenario 8

The class hierarchy is depicted in Figure D.4(c). In this scenario, `C utilizes` both *implementations* `A` and `B`. `B` contains a definition for the concrete method `m()`, while `A` declares an abstract method with the same signature. It is a conflict situation because two distinct declarations with the same signature are visible in `C`. According to the semantics of the extended language, concrete method is given precedence over abstract method in resolving conflicts and thus only the definition from `B` is made visible in `C`. At run-time, the message `c.m()` is dispatched to the method in `B`.

D.2.4 Scenario 9

Figure D.5(a) shows the relationship among the *classes* for Scenario 9, in which `C extends` *class* `A` and `utilizes` *implementation* `B`. Both `A` and `B` contain a different definition for the concrete method `m()`. It is a conflict situation in which two distinct declarations with the same signature are visible in `C`. Since both implementations are equal in precedence, the conflict cannot be resolved and an error will be raised at compile time. If it is the result of recompilation of one of the *implementations* without recompiling `C`, an ambiguity exception will be raised at run-time.

D.2.5 Scenario 10

Figure D.5(b) shows the situation in Scenario 10. This is very similar to Scenario 9, except that `A` is an *implementation* instead of a *class*. Since a *class* and an *implementation* are treated equally in code inheritance in the new semantics of the language, the outcome

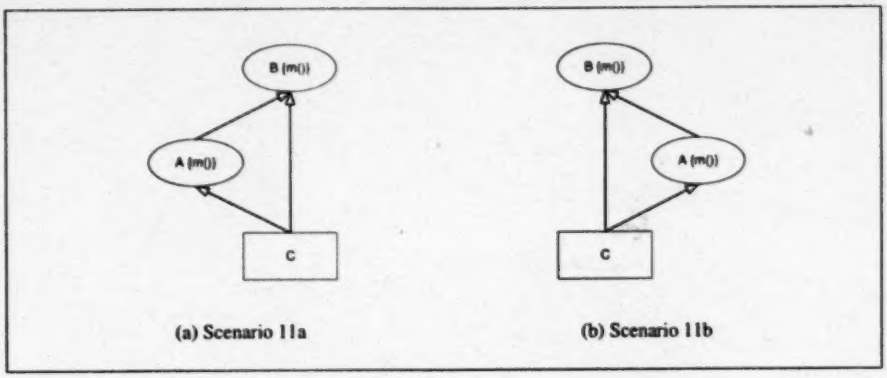


Figure D.6: Scenarios 11a and 11b

in this scenario is the same as in the previous one.

D.3 Multiple inheritance paths from the same root

D.3.1 Scenario 11

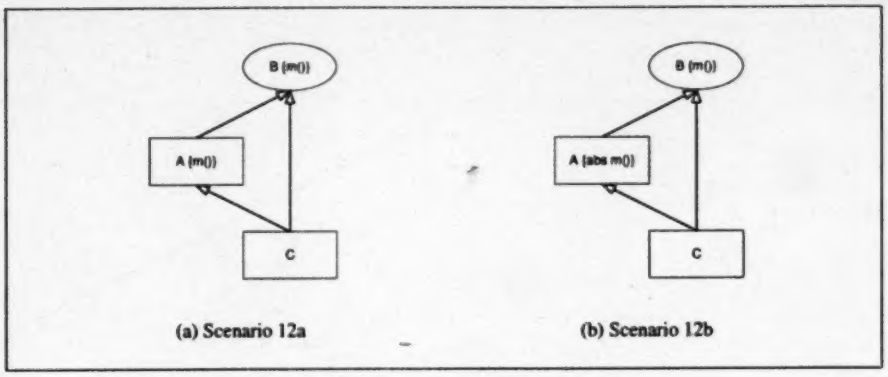
Figure D.6 shows two distinct versions of Scenario 11. The two differ only in the lexical order of the *supertypes* in the declaration line of C. However, the semantics of both the original and the extended version of Java do not attach weight to the lexical order of a *type* within the list of *supertypes*. Therefore, the two versions of this scenario will have the same outcome, independent of the lexical order of the *supertyes*.

Two distinct implementations of `m()` can be seen in C; it is a conflict situation. Since there is a partial order between A and B, the definition found in B is overridden by the one found in A according to the *relaxed multiple inheritance* approach, making only the definition found in A visible in C. Thus, the message `c.m()` sent at run-time will be dispatched to the method found in A.

D.3.2 Scenario 12

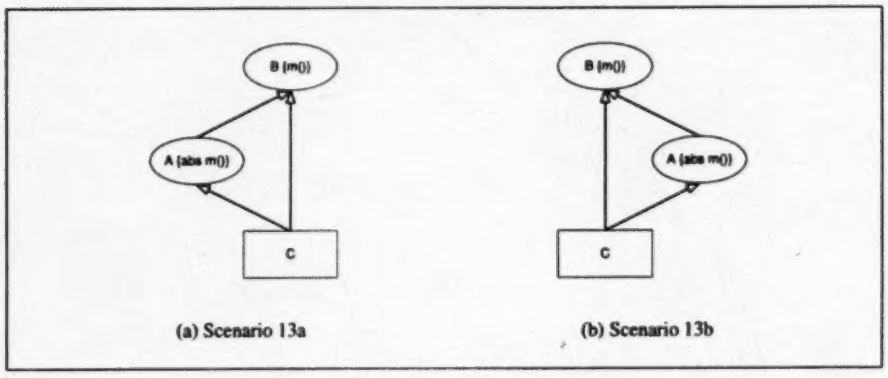
Based on the explanation found in the discussion of Scenario 11, the message `c.m()` sent at run-time in Scenario 12a (as depicted in Figure D.7(a)) will be dispatched to the method defined in A.

However, the situation is changed in Scenario 12b (as shown in Figure D.7(b)) because



(a) Scenario 12a (b) Scenario 12b

Figure D.7: Scenarios 12a and 12b



(a) Scenario 13a (b) Scenario 13b

Figure D.8: Scenarios 13a and 13b

the method declared in A is an abstract method; it is a suspension of inheritance. According to the semantics of language extension, precedence is given to a concrete method over an abstract method. Thus, the message `c.m()` sent at run-time in Scenario 12b will be dispatched to the method defined in B.

D.3.3 Scenario 13

Figure D.8 shows two versions of Scenario 13. They differ in the lexical order of the *supertypes*. As previously explained, the outcome is the same because there is no weight attached to the lexical order of a *type* within the type list.

Since the definition found in B is given precedence over the one found in A, because it is an abstract method, the definition of `m()` in B is visible in C. At run-time, the message

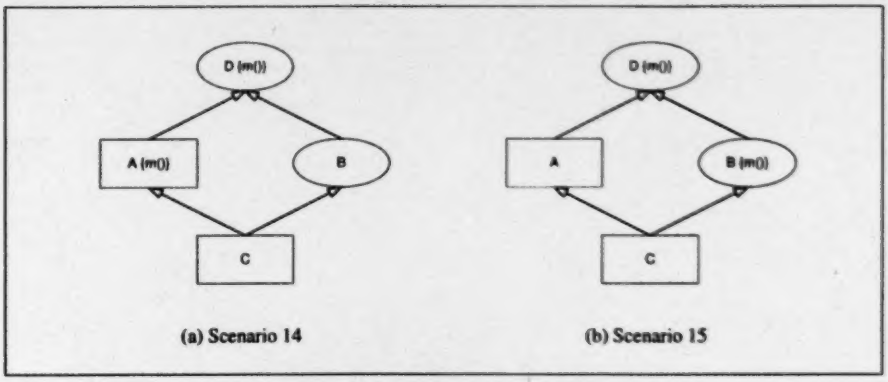


Figure D.9: Scenarios 14 and 15

`c.m()` will be dispatched to the method defined in `B`.

D.4 Diamond-shaped multiple inheritance paths

D.4.1 Scenario 14

Scenario 14, as shown in Figures D.9(a), is the situation in which `C` `extends` its *superclass* `A`. `A` has defined the method `m()` which overrides the declaration in `D` from which `A` inherits. `C` also `utilizes` *implementation* `B`, which also `extends` `D`.

The definition of `m()` in `D` is overridden by the definition in `A`, so it is hidden from `C` along this inheritance path. However, it is not overridden along the path via `B`. The situation is similar to that of Scenario 12a, as shown in Figure D.7(a). It is therefore quite natural to expect the same outcome as found in Scenario 12a for Scenario 14.

By virtue of the *relaxed multiple code inheritance* approach, the definition of `m()` visible in `C` will be the one defined in `A`. At run-time, the message `c.m()` will be dispatched to the method defined in `A`, because the definition of `m()` in `D` is farther away from `C`.

D.4.2 Scenario 15

As shown in Figure D.9(b), Scenario 15 has the same inheritance structure as that found in Scenario 14. In this scenario, a new definition of `m()` is found in `B` instead of `A`, as in Scenario 14.

Similar to Scenario 14, by virtue of the *relaxed multiple code inheritance* approach, the

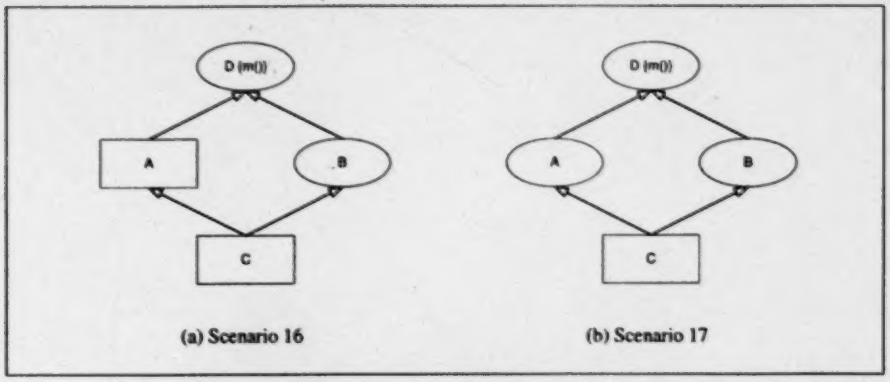


Figure D.10: Scenarios 16 and 17

definition of `m()` visible in `C` will be the one defined in `B`. At run-time, the message `c.m()` will be dispatched to the method defined in `B`, because the definition of `m()` in `D` is farther away from `C`.

D.4.3 Scenario 16

As shown in Figure D.10(a), Scenario 16 has the same inheritance structure as that found in Scenario 14. In this scenario, no new definition of `m()` is found in `A` or `B`. The definition in `D` is the definition visible in `C`. While the method is inherited from `D` via two different paths, the two inherited methods are actually the same. The new semantics does not define this situation as a conflict. At run-time, the message `c.m()` will be dispatched to the method defined in `D`, as it is the only one visible in `C`.

D.4.4 Scenario 17

Depicted in Figure D.10(b), Scenario 17 is a situation in which `C utilizes` two *implementations*, `A` and `B`. At the same time, both `A` and `B extend` another *implementation*, `D`. The definition of method `m()` in `D` is not overridden in `A` or `B`.

The definition of `m()` visible in `C` is inherited from the same source via two distinct inheritance paths. No conflict will be reported using the extended language semantics and the definition in `D` will be taken as the one visible in `C`. Thus, at run-time, the message `c.m()` will be dispatched to the method declared in `D`.

This page contains no text

Appendix E

Refactoring the `java.io` package

E.1 Result of refactoring

Class	RAF		DIS		DOS		IC		OC	
Method	**B**	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**	**A**
writeBoolean	1	0	-	-	2	2	-	-	-	2
writeByte	1	0	-	-	2	2	-	-	-	2
writeShort	2	0	-	-	4	2	-	-	-	3
writeChar	2	0	-	-	4	2	-	-	-	3
writeInt	4	0	-	-	6	2	-	-	-	5
writeLong	8	0	-	-	10	2	-	-	-	9
writeFloat	1	0	-	-	1	0	-	-	-	1
wrteDouble	1	0	-	-	1	0	-	-	-	1
sink	-	-	-	-	-	1	-	-	-	1
readFully(byte[])	1	0	1	0	-	-	-	1	-	-
readFully(byte[],int,int)	5	0	6	0	-	-	-	5	-	-
readBoolean	3	0	3	0	-	-	-	3	-	-
readByte	3	0	3	0	-	-	-	3	-	-
readUnsignedByte	3	0	3	0	-	-	-	3	-	-
readShort	4	0	5	0	-	-	-	4	-	-
readUnsignedShort	4	0	5	0	-	-	-	5	-	-
readChar	4	0	5	0	-	-	-	5	-	-
readInt	6	0	7	0	-	-	-	7	-	-
readLong	1	0	2	0	-	-	-	1	-	-
readFloat	1	0	1	0	-	-	-	1	-	-
readDouble	1	0	1	0	-	-	-	1	-	-
readUTF	1	0	1	0	-	-	-	30	-	-
static readUTF	-	-	30	1	-	-	-	-	-	-
source	-	-	-	1	-	-	-	1	-	-
Total	57	0	73	2	30	13	-	70	-	27

Legends:
RAF = RandomAccessFile
DIS = DataInputStream
DOS = DataOutputStream
IC = InputCode
OC = OutputCode
B = Before refactoring
A = After refactoring

Table E.1: Change in the number of executable statements as a result of refactoring

E.2 Source code

E.2.1 Source.java

```
package java.io;

public interface Source
{
    public int read() throws IOException;
    public int read(byte b[], int off, int len) throws IOException;
}
```

E.2.2 InputCode.java

```
package java.io;

public implementation InputCode implements DataInput, Source
{
    public Source source() {
        return this;
    }
    public void readFully(byte b[]) throws IOException {
        this.readFully(b, 0, b.length);
    }
    public void readFully(byte b[], int off, int len) throws IOException {
        int n = 0;
        while (n < len) {
            int count = this.source().read(b, off + n, len - n);
            if (count < 0)
                throw new EOFException();
            n += count;
        }
    }
    public boolean readBoolean() throws IOException {
        int ch = this.source().read();
        if (ch < 0)
            throw new EOFException();
        return (ch != 0);
    }
    public byte readByte() throws IOException {
        int ch = this.source().read();
        if (ch < 0)
            throw new EOFException();
        return (byte)(ch);
    }
    public int readUnsignedByte() throws IOException {
        int ch = this.source().read();
        if (ch < 0)
            throw new EOFException();
        return ch;
    }
    public short readShort() throws IOException {
        int ch1 = this.source().read();
        int ch2 = this.source().read();
        if ((ch1 | ch2) < 0)
            throw new EOFException();
        return (short)((ch1 << 8) + (ch2 << 0));
    }
    public int readUnsignedShort() throws IOException {
        Source in = this.source();
        int ch1 = in.read();
        int ch2 = in.read();
        if ((ch1 | ch2) < 0)
            throw new EOFException();
        return (ch1 << 8) + (ch2 << 0);
    }
```

```
    public char readChar() throws IOException {
        Source in = this.source();
        int ch1 = in.read();
        int ch2 = in.read();
        if ((ch1 | ch2) < 0)
            throw new EOFException();
        return (char)((ch1 << 8) + (ch2 << 0));
    }
    public int readInt() throws IOException {
        Source in = this.source();
        int ch1 = in.read();
        int ch2 = in.read();
        int ch3 = in.read();
        int ch4 = in.read();
        if ((ch1 | ch2 | ch3 | ch4) < 0)
            throw new EOFException();
        return ((ch1 << 24) + (ch2 << 16) + (ch3 << 8) + (ch4 << 0));
    }
    public long readLong() throws IOException {
        return ((long)(this.readInt()) << 32) + (this.readInt() & 0xFFFFFFFFL);
    }
    public float readFloat() throws IOException {
        return Float.intBitsToFloat(this.readInt());
    }
    public double readDouble() throws IOException {
        return Double.longBitsToDouble(this.readLong());
    }
    String readUTF() throws IOException {
        int utflen = this.readUnsignedShort();
        char str[] = new char[utflen];
        int count = 0;
        int strlen = 0;
        while (count < utflen) {
            int c = this.readUnsignedByte();
            int char2, char3;
            switch (c >> 4) {
                case 0: case 1: case 2: case 3: case 4: case 5: case 6: case 7:
                    count++;
                    str[strlen++] = (char)c;
                    break;
                case 12: case 13:
                    count += 2;
                    if (count > utflen)
                        throw new UTFDataFormatException();
                    char2 = this.readUnsignedByte();
                    if ((char2 & 0xC0) != 0x80)
                        throw new UTFDataFormatException();
                    str[strlen++] = (char)(((c & 0x1F) << 6) | (char2 & 0x3F));
                    break;
                case 14:
                    count += 3;
                    if (count > utflen)
                        throw new UTFDataFormatException();
                    char2 = this.readUnsignedByte();
                    char3 = this.readUnsignedByte();
                    if (((char2 & 0xC0) != 0x80) || ((char3 & 0xC0) != 0x80))
                        throw new UTFDataFormatException();
                    str[strlen++] = (char)(((c & 0x0F) << 12) |
                                           ((char2 & 0x3F) << 6) |
                                           ((char3 & 0x3F) << 0));
                    break;
                default:
                    throw new UTFDataFormatException();
                }
            }
        return new String(str, 0, strlen);
    }
}
```

E.2.3 Sink.java

```
package java.io;

public interface Sink
{
    public void write(int b) throws IOException;
}
```

E.2.4 OutputCode.java

```
package java.io;

public implementation OutputCode implements DataOutput, Sink
{
    public Sink sink(){
        return this;
    }
    public void writeBoolean(boolean v) throws IOException {
        Sink out = this.sink();
        out.write(v ? 1 : 0);
    }
    public void writeByte(int v) throws IOException {
        Sink out = this.sink();
        out.write(v);
    }
    public void writeShort(int v) throws IOException {
        Sink out = this.sink();
        out.write((v >>> 8) & 0xFF);
        out.write((v >>> 0) & 0xFF);
    }
    public void writeChar(int v) throws IOException {
        Sink out = this.sink();
        out.write((v >>> 8) & 0xFF);
        out.write((v >>> 0) & 0xFF);
    }
    public void writeInt(int v) throws IOException {
        Sink out = this.sink();
        out.write((v >>> 24) & 0xFF);
        out.write((v >>> 16) & 0xFF);
        out.write((v >>>  8) & 0xFF);
        out.write((v >>>  0) & 0xFF);
    }
    public void writeLong(long v) throws IOException {
        Sink out = this.sink();
        out.write((int)(v >>> 56) & 0xFF);
        out.write((int)(v >>> 48) & 0xFF);
        out.write((int)(v >>> 40) & 0xFF);
        out.write((int)(v >>> 32) & 0xFF);
        out.write((int)(v >>> 24) & 0xFF);
        out.write((int)(v >>> 16) & 0xFF);
        out.write((int)(v >>>  8) & 0xFF);
        out.write((int)(v >>>  0) & 0xFF);
    }
    public void writeFloat(float v) throws IOException {
        this.writeInt(Float.floatToIntBits(v));
    }
    public void writeDouble(double v) throws IOException {
        this.writeLong(Double.doubleToLongBits(v));
    }
}
```

E.2.5 RandomAccessFile.java

```
package java.io;

public class RandomAccessFile utilizes InputCode, OutputCode
{
    public RandomAccessFile(String name, String mode) throws FileNotFoundException
    {
        **** code not modified ****
    }
    public RandomAccessFile(File file, String mode) throws IOException {
        **** code not modified ****
    }
    public final FileDescriptor getFD() throws IOException {
        **** code not modified ****
    }
    private native void open(String name, boolean writeable) throws FileNotFoundException;

    public native int read() throws IOException;

    private native int readBytes(byte b[], int off, int len) throws IOException;

    public int read(byte b[], int off, int len) throws IOException {
        **** code not modified ****
    }
    public int read(byte b[]) throws IOException {
        **** code not modified ****
    }
    public int skipBytes(int n) throws IOException {
        **** code not modified ****
    }
    public native void write(int b) throws IOException;

    private native void writeBytes(byte b[], int off, int len) throws IOException;

    public void write(byte b[]) throws IOException {
        **** code not modified ****
    }
    public void write(byte b[], int off, int len) throws IOException {
        **** code not modified ****
    }
    public native long getFilePointer() throws IOException;

    public native void seek(long pos) throws IOException;

    public native long length() throws IOException;

    public native void setLength(long newLength) throws IOException;

    public native void close() throws IOException;

    public final String readLine() throws IOException {
        **** code not modified ****
    }
    public final void writeBytes(String s) throws IOException {
        **** code not modified ****
    }
    public final void writeChars(String s) throws IOException {
        **** code not modified ****
    }
    public final void writeUTF(String str) throws IOException {
        **** code not modified ****
    }
}
```

E.2.6 DataOutputStream.java

```
package java.io;

public class DataOutputStream extends FilterOutputStream utilizes OutputCode
{
    public DataOutputStream(OutputStream out) {
        ****  code not modified  ****
    }
    private void incCount(int value) {
        ****  code not modified  ****
    }
    public synchronized void write(int b) throws IOException {
        ****  code not modified  ****
    }
    public synchronized void write(byte b[], int off, int len) throws IOException
    {
        ****  code not modified  ****
    }
    public void flush() throws IOException {
        ****  code not modified  ****
    }
    public final void writeBoolean(boolean v) throws IOException {
        super(OutputCode).write(v ? 1 : 0);
        incCount(1);
    }
    public final void writeByte(int v) throws IOException {
        super(OutputCode).write(v);
        incCount(1);
    }
    public final void writeShort(int v) throws IOException {
        super(OutputCode).writeShort(v);
        incCount(2);
    }
    public final void writeChar(int v) throws IOException {
        super(OutputCode).writeChar(v);
        incCount(2);
    }
    public final void writeInt(int v) throws IOException {
        super(OutputCode).writeInt(v);
        incCount(4);
    }
    public final void writeLong(long v) throws IOException {
        super(OutputCode).writeLong(v);
        incCount(8);
    }
    public final void writeBytes(String s) throws IOException {
        ****  code not modified  ****
    }
    public final void writeChars(String s) throws IOException {
        ****  code not modified  ****
    }
    public final void writeUTF(String str) throws IOException {
        ****  code not modified  ****
    }
    public final int size() {
        ****  code not modified  ****
    }
    //  overriding the method declared in OutputCode
    public Sink sink(){
        return this.out;
    }
}
```

E.2.7 DataInputStream.java

```
package java.io;

public class DataInputStream extends FilterInputStream utilizes InputCode
{
    public DataInputStream(InputStream in) {
        ****  code not modified  ****
    }
    public final int read(byte b[]) throws IOException {
        ****  code not modified  ****
    }
    public final int read(byte b[], int off, int len) throws IOException {
        ****  code not modified  ****
    }
    public final int skipBytes(int n) throws IOException {
        ****  code not modified  ****
    }
    public final String readLine() throws IOException {
        ****  code not modified  ****
    }
    public final static String readUTF(DataInput in) throws IOException {
        return in.readUTF();
    }
    //  overriding the method declared in InputCode
    public Source source(){
        return this.in;
    }
}
```